University of Alberta

MedVis: A Real-Time Immersive Visualization Environment for the Exploration of Medical Volumetric Data

by

Rui Shen

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-33350-1
Our file Notre référence
ISBN: 978-0-494-33350-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

An attempt at visualizing the Fourth Dimension: Take a point, stretch it into a line, curl it into a circle, twist it into a sphere, and punch through the sphere.

– Albert Einstein, 1879-1955, German-born American physicist.

To my beloved parents.

Abstract

This thesis introduces the Medical Visualizer (MedVis), a real-time visualization system for analyzing medical volumetric data in various virtual environments, such as parallax barrier autostereoscopic displays, dual-projector screens and immersive environments such as the CAVE. Direct volume rendering is used for visualizing the details of medical volumetric data sets without intermediate geometric representations. By interactively manipulating the color and transparency functions, radiologists can either inspect the data set as a whole or focus on a specific region. In this system, 3D texture hardware is employed to accelerate the rendering process. The system is designed to be platform independent, as all virtual reality functions are separated from kernel functions. Due to the modular design, it can be easily extended to other virtual environments, and new functions can be incorporated rapidly.

Acknowledgements

First, I would like to thank my supervisor Dr. Pierre Boulanger for his guidance and support throughout my thesis research. Next, many thanks to Dr. Michelle Noga for initiating the medical visualization project and providing the volumetric medical data. Special thanks to Ryan Hung for his help in prototyping the visualization pipeline in AVS/Express. Special thanks also go to Jacques-Andre Boulay for his help on the configuration of VR Juggler for the CAVE. Finally, I would like to give my thanks to Guowei Wu and Cheng Lei for their help on GPU programming, and Xingdong Yang for always providing useful feedback when I discuss my ideas with him.

Table of Contents

1 Introduction 1
- 1.1 Visualization . . . 1
- 1.2 Medical Visualization . . . 1
- 1.3 Thesis Contributions . . . 2
- 1.4 Thesis Overview . . . 3

2 Related Work 4
- 2.1 Medical Data Acquisition . . . 4
  - 2.1.1 Computed Tomography . . . 4
  - 2.1.2 Magnetic Resonance Imaging . . . 4
  - 2.1.3 Other Acquisition Technologies . . . 5
- 2.2 Volume Rendering . . . 5
  - 2.2.1 Object-Order Volume Rendering . . . 6
  - 2.2.2 Image-Order Volume Rendering . . . 10
  - 2.2.3 Domain Volume Rendering . . . 13
  - 2.2.4 Hardware-Accelerated Volume Rendering . . . 14
  - 2.2.5 Critical Review of Volume Rendering Algorithms . . . 18
- 2.3 Virtual Environments . . . 18
  - 2.3.1 Projection-Based VR Systems . . . 19
  - 2.3.2 Monitor-Based VR Systems . . . 20
  - 2.3.3 HMD-Based VR Systems . . . 21
- 2.4 Visualization in Virtual Environments . . . 22
  - 2.4.1 VR Systems for General-Purpose Visualization . . . 22
  - 2.4.2 VR Systems for Medical Visualization . . . 23
- 2.5 Summary . . . 25

3 System Design and Implementation 27
- 3.1 Overview . . . 27
- 3.2 Rendering Algorithms . . . 27
  - 3.2.1 Direct Volume Rendering . . . 28
  - 3.2.2 Stereo Rendering . . . 37
- 3.3 Hardware Setup . . . 38
  - 3.3.1 Desktop Version . . . 38
  - 3.3.2 Immersive CAVE Version . . . 39
- 3.4 Interaction Modalities . . . 40
  - 3.4.1 Desktop Interaction . . . 41
  - 3.4.2 CAVE Interaction . . . 42
- 3.5 Software Architecture . . . 44
  - 3.5.1 Toolkits Used . . . 44
  - 3.5.2 MedVis Kernel Module . . . 46
  - 3.5.3 MedVis Desktop Interface Module . . . 48
  - 3.5.4 MedVis CAVE Interface Module . . . 50

4 Performance and Results 53
4.1 System Performance for Rendering 53
4.2 Results for the Desktop Version 58
4.3 Results for the CAVE Version 59

5 Conclusion and Future Work 61
5.1 Future Work 62

Bibliography 63

A Cg Code 71
A.1 Vertex Shader Code 71
A.2 Fragment Shader Code 72

B GPU-Based Volume Rendering C++ Code 73
B.1 vtkVolumeTextureMapper3DCg.h 73
B.2 vtkVolumeTextureMapper3DCg.cxx 74

List of Tables

3.1 The computation of the intersection points. 33

4.1 The rendering times using different acceleration techniques. 53

List of Figures

2.1 Shear-warp for orthographic projections. 10
2.2 Shear-warp for perspective projections. 10
2.3 Discrete ray representation. 13
2.4 Discrete ray tracing algorithm. 13
2.5 Volume rendering using 3D texture mapping. 15

3.1 The comparison of classification first and interpolation first. 29
3.2 The intersection cases between a proxy plane and the volume bounding box. 32
3.3 The traversal of the bounding box edges. 33
3.4 The dataflow between the CPU and the GPU. 35
3.5 Stereo rendering. 37
3.6 MedVis desktop hardware setup. 39
3.7 MedVis CAVE hardware setup. 40
3.8 Microsoft force feedback joystick. 40
3.9 InterSense IS-900. 40
3.10 The desktop control panel. 41
3.11 Re-slicing along the current viewing direction. 43
3.12 The CAVE control panel. 43
3.13 MedVis's visualization pipeline. 44
3.14 The vtkDICOMVolume class. 47
3.15 The histogram classes. 48
3.16 The vtkStereoRenderWindowInteractor class. 49
3.17 The vtkGtkStereoRenderWindowInteractor class. 49
3.18 The vtkGtkColorHistogram class. 50
3.19 The vtkGtkSliceViewer2 class. 50
3.20 The vtkVRJApp class. 50
3.21 The vtkVRJDICOMVolume class. 51
3.22 The vtkVRJTransferFunctionEditor class. 52

4.1 The comparison of the rendering speeds using different acceleration techniques. 54
4.2 Volume rendering results of a CT-scanned pelvic region in MedVis. . 55
4.3 The rendering speed of MedVis's GPU-based algorithm with different sample intervals. 56
4.4 Volume rendering results of a CT-scanned pelvic region in MedVis with different sample intervals. 57
4.5 Stereo volume rendering of a CT-scanned abdomen in the MedVis desktop version. 59
4.6 Volume rendering of an MRI-scanned heart in the MedVis CAVE version. 60

Chapter 1

Introduction

1.1 Visualization

Visualization is the process of converting raw numbers into representations that can be easily interpreted by the human visual system (HVS) [84]. Exploiting the human visual system for interpreting the data, visualization has become an essential tool for solving scientific problems. While computers excel at simulations, data filtering, and data reduction, humans are experts at using their highly developed pattern-recognition skills to locate regions of interest, features, and anomalies [65] [104]. When visualized data are volumetric data, a specific term is used: *volume visualization*. It is the process of projecting a multidimensional data set onto a two-dimensional image plane for the purpose of gaining an understanding of the structure (or lack of) contained within the volumetric data [26]. Volume visualization is used in many fields, such as medicine, architecture, archaeology and engineering.

1.2 Medical Visualization

Medical imaging modalities such as *Computed Tomography* (CT) and *Magnetic Resonance Imaging* (MRI) produce high-quality 3D data that radiologists use to diagnose various health problems of patients. However, because of technological limitations, it is still commonplace for radiologists to observe volumetric data as a set of slices printed on films viewed in front of a white diffusing light source. Even though this practice is well accepted in the medical community, it only utilizes a fraction of the data provided by the imaging systems. The common rational for using such system, aside from its cost, is that the quality of the contrast in film

is far superior to normal CRT screens, and more importantly that the data set is not manipulated by computer processing leaving image interpretation to the far superior human visual system. The last requirement is key as radiologists want to observe the data set without prior interpretations by a computer that may hide important structures that only human can interpret as important. Several systems, from non-commercial software (*e.g.*, ParaView [49]) to commercial products (*e.g.*, VolView [48] and AVS/Express [1]), have been developed for this purpose. However, the visualization and manipulation stay in the 2D space, *i.e.*, users can only see the volumetric data as a projected 2D image on a screen and the feedback of an operation on the data is still a 2D image. Although volume rendering is supported in current systems, radiologists still cannot directly examine 3D volumes in stereo. This problem can be resolved by using virtual reality (VR) techniques. The usefulness of VR in scientific visualization has been discussed abundantly in the literature [11] [104]. In many ways, the major impact of virtual reality technologies on scientific visualization is in providing a "real-time" intuitive interface for exploring data while facilitating the use of scientific visualization [11]. In particular, the effectiveness of VR technology in medicine has also been proven in many applications described in [82] and [122]. Hence, to assist existing diagnostic procedure by creating a new insight through 3D visual representation, we have developed the *Medical Visualizer* (MedVis), a medical visualization system that provides real-time high-quality volume rendering and interaction with 3D medical data in virtual environments (VEs).

1.3 Thesis Contributions

This thesis makes contributions to the following aspects of medical volumetric data visualization:

1. Developing a cross-platform medical data visualization system capable of dealing with various display modalities. Radiologists can use this system to interactively explore medical volumetric data in stereo in various non-immersive or immersive virtual environments.

2. Developing new acceleration techniques of volume rendering using general-purpose GPU. Remarkable speedups compared to traditional techniques are observed from real experiments.

3. Object-oriented programming paradigm and modular design concept are employed throughout the development of MedVis, which maximizes the extensibility and portability of the system.

1.4 Thesis Overview

The thesis is organized as follows. Chapter 2 introduces medical data acquisition techniques, and reviews volume rendering methods, virtual reality systems and the developments of visualization systems in virtual environments. Chapter 3 describes the design and implementation of MedVis, including the hardware and software architecture, rendering algorithms and interaction schemes. Chapter 4 presents the results of MedVis and discusses its performance, especially the performance of the proposed rendering algorithms. Chapter 5 concludes the thesis and presents future research directions.

Chapter 2

Related Work

2.1 Medical Data Acquisition

Visualization quality depends on the quality of the input data. It is almost impossible to generate effective visualization while the acquired data are inaccurate or insufficient. Given the significance of data acquisition in medicine, this section gives a brief introduction to medical imaging modalities, which are capable of producing high-quality 3D data composed of 2D image slices.

2.1.1 Computed Tomography

Computed Tomography (CT) is a medical imaging modality invented by Hounsfield in 1972 that uses X-ray to generate cross sections of objects. A CT scanner is a X-ray instrument capable of digitizing full bodies in 3D at very high precision (~1mm) and speed [119]. Images are acquired from a rotary X-ray fan source revolving around the patient. The X-ray fan source is then digitized by a circular network of solid state X-ray detectors, to which a tomography reconstruction algorithm is applied. The values in the CT data set correspond to the average density values of voxels inside the human body. Dense objects (*e.g.*, bones) tend to absorb more X-rays than less dense objects (*e.g.*, muscle). The absorption characteristic of each voxel is calibrated in Hounsfield units (HU), a measurement normalized with respect to the attenuation of X-ray in water.

2.1.2 Magnetic Resonance Imaging

Magnetic Resonance Imaging (MRI) is another medical imaging modality developed by Lauterbur and Mansfield in the 1970s. An MRI scanner is composed of a large magnet, a microwave transmitter, a microwave antenna, and several electronic

components that decode the signal and reconstruct cross-sectional images from the data [119]. Unlike CT, which uses ironizing radiation, MRI measures the relaxation properties of excited hydrogen nuclei, *i.e.*, it is based on the measurement of radio frequency of electromagnetic waves emitted by a nucleus spin when returning to its equilibrium state from the excited state produced by a microwave emitter [45]. Nuclei within different tissues emit signals of different frequencies, which depend on water density present in the tissue. *Functional MRI* (fMRI) is a new type of MRI modality that records both the patient's anatomy and the physiological functions of the tissues being studied. By measuring the oxygenation value of blood, fMRI allows imaging regions with high consumption of oxygen, which is a characteristic of higher metabolic activities.

2.1.3 Other Acquisition Technologies

In addition to CT and MRI, there are two other types of medical imaging modalities, which are used every day in hospitals. *Nuclear medicine* techniques, including *Single Photon Emission Computed Tomography* (SPECT) and *Positron Emission Tomography* (PET), use the decay of injected radioactive isotopes to image the distribution of the isotope as a function of time and space [45]. Other modalities such as *Ultrasonography* work in ways similar to CT, but instead of X-rays, ultrasound is employed to illuminate the object. The echo information is recorded to determine the type and position of the object. In most cases, all those sensors produce volumetric data, where the scalar value at each volume element (voxel) depends on a particular tissue property.

2.2 Volume Rendering

Volume rendering deals with how a 3D volume is projected onto the view plane to form a 2D image. It has been broadly used in medical applications for planning treatments [61] and to help in diagnosis [38]. At present, there are two main ways to visualize volumetric data: surface rendering and direct volume rendering (DVR). Surface rendering is also referred to as indirect volume rendering, because it is based on one or more scalar thresholds that are used to compute iso-surfaces of the volumetric data that are then polygonized and rendered using normal polygon rendering schemes. Iso-surfaces can be generated by using techniques such as

marching cubes [62]. In contrast to iso-surface methods, which analyze volumetric data in the local neighborhoods, Arvo and Novins [4] propose an iso-contour method, which operates on the continuous image space to extract curves of constant intensity. However, surface rendering depends on an existence assumption that a set of iso-surfaces exists, and on a fidelity assumption that with the infinitely thin surface the polygon mesh models the true object structures at reasonable fidelity [66]. These two assumptions can hardly be met simultaneously in practice. In contrast, direct volume rendering bypasses the intermediate geometric representation and directly renders the volumetric data set based on its scalar values alone. Color mapping and transparency schemes are commonly employed to enhance the visual contrast between different materials. According to Kaufman [46], volume rendering approaches can be classified into three categories: object-order, image-order and domain methods.

2.2.1 Object-Order Volume Rendering

The object-order approaches evaluate the final pixel values in a back-to-front or front-to-back fashion, *i.e.*, the scalar values in each voxel[1] are accumulated along the view direction.

Drebin *et al.* [25] propose a material-based rendering method. The volume is assumed to be an image stack. Five steps are performed to generate the final image: classification, matting, surface extraction, shading and projection. In the classification step, for each voxel, the percentage of each material is estimated using probabilistic classifiers. The classification transforms the initial volume into a set of material percentage volumes. This set of volumes is used to calculate the properties of each voxel, such as color[2]. Next, *matte volumes* are combined with the classified volume using *spatial set operations*, roughly like boolean operations in constructive solid geometry (CSG) [79], to remove undesired regions and/or adjust the percentage of a certain material. Surfaces, *i.e.*, boundaries between different materials are extracted from a so-called ρ volume. The parameter ρ, the value of which actually can be assigned arbitrarily, characterizes the density of a material. While two close ρ's blur the boundary between two materials, two diverse ρ's intensify the boundary. After every material is assigned a ρ value, the density D of a voxel is computed

[1]The region of constant value that surrounds each sample in zero-order interpolation [46].
[2]The word *color* when used separately throughout this thesis refers to RGBA color.

by a function proportional to ρ. The surface normals and magnitudes (strengths) derived from the gradient ∇D, which indicates the sharpness of the density transitions between voxels, are used in the next step for shading. In the shading step, light rays are cast through the volume from back to front. Let I and I' denote the intensities of the incoming ray and outgoing ray respectively, then the resulting color of a voxel is

$$I' = C + (1 - \alpha_c)I \tag{2.1}$$

where α_c is the alpha component of the RGBA color C of the current voxel. If one takes surface shading into account, C is a function of the surface normal, the surface strength, the surface diffuse color and the light source color. This shaded volume is geometrically transformed and resampled to lie along the view direction. To form the final image, an orthographic projection is performed using the following equation:

$$I_z = C_z + (1 - \alpha_c)I_{z+1} \tag{2.2}$$

where I_z and C_z are the accumulated image and color of the $z'th$ plane respectively. The initial image is set to black and I_0 is the final image. Perspective projections can also be easily applied by scaling the images in the x-y plane according to the eye's z coordinate.

Many of the techniques initiated in this paper [25] influence the formation of the structure of current volume rendering pipelines. This method is suitable for high-resolution data sets, but for low-resolution data sets, the constant-variation assumption of the scalar field in each voxel causes discontinuities between voxels. To generate smoother images for small data sets, Upson [103] proposes an algorithm called V-buffer, which uses a higher-order interpolation technique than the trilinear interpolation to approximate the variation of a voxel's scalar field. However, in both methods, each voxel corresponds to only one projected position on the view plane (*i.e.*, one pixel or several adjacent pixels). As stated in [95], this kind of projection is simple and fast, but often yields image artifacts due to the discrete selection of the projected image pixel(s). Furthermore, the rendering speed is low because of many complicated operations to be computed in shading and projection.

Westover [111] [112] addresses this problem by distributing the contributions of one voxel into a region of image pixels and carrying out such operations using

a lookup table. This volume rendering method is called splatting. The main idea is to project a basic function (*e.g.*, a Gaussian kernel) onto the view plane. This algorithm consists of four major steps: view transformation, shading, reconstruction and visibility test. The most important step is reconstruction, where a kernel convolves with the volume to determine the image space footprint (or splat) of each voxel, *i.e.*, the image pixels that a voxel affects. Normally, two phases of reconstruction are needed: volume space reconstruction and image space reconstruction. The first phase reconstructs the discrete input voxels into a 3D continuous function, whereas the second phase reconstructs discrete intermediate image samples into continuous pixels. For object-order methods, voxels are directly mapped onto the final image and the footprint function is a continuous function, therefore, there is no need to involve two individual reconstruction phases. Instead, during the volume space reconstruction, the kernel is integrated along the view direction to generate the final image pixels directly. The footprint function is then defined by:

$$F(x,y) = \int_{-\infty}^{+\infty} h_v(x,y,w)dw \tag{2.3}$$

where (x,y) is the displacement of a pixel from the center of the shaded voxel's view-plane projection; $h_v()$ is the volume reconstruction kernel; and w is the kernel's w-coordinate. This function is precomputed and the results are stored in a generic footprint table. For orthographic views, the footprint function $F(x,y)$ of each voxel is the same except with an image space offset as in [112]. However, for perspective views, the function needs to be evaluated for every voxel separately, reducing the algorithm's speed significantly.

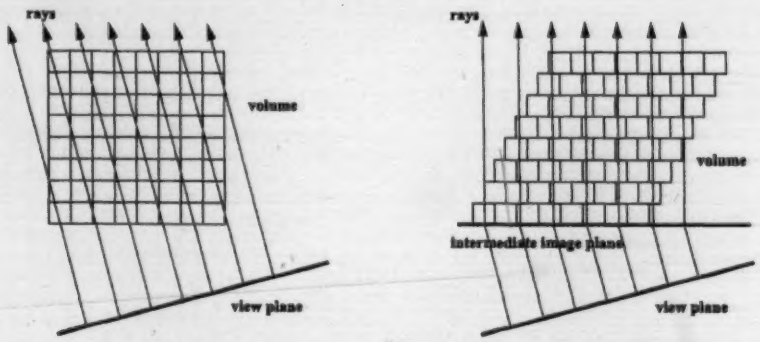
To attain a higher frame rate, Laur and Hanrahan [58] introduce hierarchical splatting. The volume is represented in a pyramidal manner and footprints of different sizes are used in different levels of the pyramid. During motion low-resolution volume representation is used, and once the motion stops a progressive refinement is applied.

Another problem with the original splatting algorithm is that the voxel kernels are integrated within the volume slices that are most parallel to the view plane, hence disturbing popping artifacts occur when the views are animated. To deal with this drawback, Mueller *et al.* [68] propose a new image-aligned splatting algorithm using slicing slabs oriented parallel to the view plane. These slabs are of certain width and only the contributions of the voxel kernels within the current

slab are added to the frame buffer. Another major advantage of this new splatting algorithm is that based on an occlusion map, only the kernels of the voxels visible in the final image are projected, which significantly speed up the algorithm.

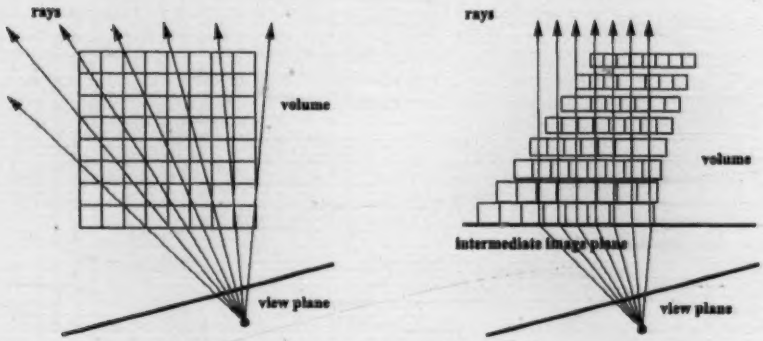
While the resampling (or reconstruction) in splatting algorithms is view-dependent, Lacroute and Levoy [56] introduce a very efficient algorithm called shear-warp, where the complications associated to resampling for arbitrary perspective views are alleviated. The input volume is composed of image slices, and it is first transformed to a sheared object space, where the viewing rays are perpendicular to the slices. For orthographic projections, the transformation is only a translation based on the slice's z-coordinate, as shown at Figure 2.1. For perspective projections, the transformation is a translation plus a uniform scaling along the volume's z-axis, as shown at Figure 2.2. The sheared slices are resampled and composited from front to back to form an intermediate image, which is then warped (resampled) to get the final image. To take advantage of the fact that the scanlines of the pixels in the intermediate image are parallel to the scanlines of the voxels in the volume data, the object-order compositing is performed in scanline order. Run-length encoded *volume scanlines* are used to skip transparent voxels and run-length encoding of *image scanlines* is used to skip occluded voxels. Run-length encoding of the volume is possible when the opacity transfer function is known in advance, however, when the transfer function needs to be changed interactively, a different acceleration scheme must be used. The opacity of a voxel is represented by a function of several precomputable parameters, such as intensity and gradient. A min-max octree (described in [114]) is applied to the volume to get the extrema of the parameters in every subvolume, and then a multi-dimensional summed-area lookup table (described in [17]) is applied to the rectangular region bounded by these extrema in the feature space to determine transparent regions. Two volume scanlines are traversed simultaneously to accelerate the compositing process, where opaque and visible voxels are projected onto the current image scanline using a bilinear interpolation technique. This algorithm can be parallelized by distributing the intermediate image scanlines to multiple processors. Schulze and Lang [92] extend the parallel architecture for perspective projections in such a way that the compositing process is distributed among remote computers using message passing interface (MPI) programming environment and the final 2D warp is performed on a single computer using graphics hardware. Although the rendering is fast, one

drawback is that three copies of the input volume, *i.e.*, the octree, the summed-area table and the original voxels, need to be maintained.



(a) Object space. (b) Sheared object space.

Figure 2.1: Shear-warp for orthographic projections.



(a) Object space. (b) Sheared object space.

Figure 2.2: Shear-warp for perspective projections.

2.2.2 Image-Order Volume Rendering

Image-order volume rendering approach is also called ray casting or ray tracing. The basic idea is that rays are cast from each pixel on the final image into the volume and the pixel values are determined by compositing the scalar values encountered along the rays with some ray function.

Like surface rendering, image-order volume rendering can also display iso-surfaces, but the difference is that in direct volume rendering shading is applied directly on the voxels to form the final image and hence small features are pre-

served. Levoy [59] proposes a pipeline for rendering iso-surfaces. The shading and classification procedures are separated in this pipeline. The output of the shading procedure is an RGB color volume and the output of the classification procedure is a volumetric opacity map. To avoid artifacts caused by a single threshold or window in traditional iso-surface methods, different opacities are assigned to the voxels with scalar values close or equal to the given threshold. During rendering, the volume is sampled at evenly-spaced locations along each ray and the color at each sample location is computed using trilinear interpolation. Parker *et al.* [72] apply ray casting for the first time to render iso-surfaces from very large data sets in an interactive system. Several optimization techniques are used to accelerate the performance. First, the volume is represented as "bricks" to improve data cache locality. Second, an octree-like spatial hierarchical data structure is employed to accelerate voxel traversing. Third, the image space is divided into tiles, and multiple processors are employed, where each processor operates on one or more of the tiles. When running on a 64-processor machine, a 1GB data set is rendered at an image resolution of 512x512 resolution at about 10 frames per second. However, due to random accessibility of the volume required by ray casting methods, the whole data set needs to be duplicated on every processor.

In addition to iso-surface, image-order volume rendering can display the volume as a whole. Kajiya and Von Herzen [44] apply ray casting to volume render natural phenomena, such as clouds, by solving a scattering equation. A major drawback of ray casting is its high complexity because for every ray the whole volume needs to be traversed once. One common optimization technique is early ray termination, which stops tracing a ray when the accumulated opacity along that ray reaches a pre-defined threshold (usually fully opaque). Levoy [60] determines the last sample location along a ray as the position with no significant change of the color of the ray. Let $C_{in}(u;U)$ and $\alpha_{in}(u;U)$ be the RGB color and opacity of the ray u before and after processing the sample U respectively, and let $C(U)$ and $\alpha(U)$ be the RGB color and opacity of the sample respectively. Then the RGB color $C_{out}(u;U)$ and the opacity $\alpha_{out}(u;U)$ of the ray u after processing the sample U are calculated by:

$$C_{out}(u;U) = \frac{C_{in}(u;U)\alpha_{in}(u;U) + C(U)\alpha(U)(1-\alpha_{in}(u;U))}{\alpha_{out}(u;U)} \tag{2.4}$$

and

$$\alpha_{out}(u;U) = \alpha_{in}(u;U) + \alpha(U)(1 - \alpha_{in}(u;U)) \tag{2.5}$$

The RGB color and opacity are accumulated from front to back. A significant RGB color change occurs when $C_{out}(u;U) - C_{in}(u;U) > \varepsilon$ for some small $\varepsilon > 0$. Hence, a ray terminates when $\alpha_{out}(u;U) > 1 - \varepsilon$ for the first time. Another common optimization technique is empty space skipping, which accelerates the traversal of empty voxels (*i.e.*, voxels with zero-opacity). Levoy [60] represents volumetric data as a pyramid (an octree-like structure). Each cell at each level of the pyramid carries a binary value. A higher level cell carries a zero when its eight children all contain zeros. In the lowest level, where a cell corresponds to a voxel, a cell contains zero if the associated voxel is empty. When ray tracing is performed, empty regions can be effectively bypassed, because whenever an empty cell is encountered, its descendants are not traversed.

Danskin and Hanrahan [22] extend Levoy's work by a homogeneity-acceleration and a β-acceleration. The homogeneity-acceleration employs a $range_{27}$ pyramid. Each cell of the pyramid contains a Manhattan distance, which measures the homogeneity at that cell. A smaller distance indicates that the cell is more homogeneous, and hence a larger ray sampling interval can be used; a larger distance indicates that the cell is more heterogeneous, and hence a smaller sampling interval should be used. The fundamental idea of the β-acceleration is that as the pixel opacity (*i.e.*, the β-distance) along a ray accumulates from front to back, less light travels back to the eye, therefore, fewer ray samples need to be taken without significant changes to the final image quality. As for the 4D cases, Yagel and Shi [118] use space-leaping for volume animation, which exploits coherency between consecutive images to shorten the paths that rays take through the volume.

In the previous image-order methods, continuous rays are sampled at uniform or jittered intervals. Alternatively, rays can be represented as discrete voxels [115] [117] as shown in Figure 2.3. A discrete ray can be either 6-connected or 26-connected. A ray is 6-connected if any two adjacent voxels share only a face, whereas any two adjacent voxels of a 26-connected ray share a face, an edge or a vertex. Discrete ray tracing, or 3D Raster Ray Tracing (RRT), is utilized by Yagel and Kaufman [116] in a template-based ray casting algorithm for orthographic projections. This algorithm is based on the observation that for orthographic discrete ray casting all rays have the same form and this coherency between rays can be ex-

ploited to simplify the volume traversal process. Like the shear-warp method [56], the volume is first projected onto an intermediate plane and then the intermediate image is warped to form the final image. The difference is that the intermediate image is generated in a pixel-by-pixel manner. Three phases are involved: initialization, ray casting and 2D mapping. The principle of the algorithm is illustrated at Figure 2.4. During initialization, a base plane parallel to one of the volume's faces is determined, and the volume is projected onto the base plane. The form of the parallel rays is computed and stored in a template. Here, 26-connected rays instead of 6-connected rays are used, because every voxel in the volume is visited exactly once. Next, all the rays are generated from this template and cast from the base plane into the volume. The rays are traced by uniform steps and the voxel coordinates at each step can be computed efficiently using the template. Finally, the image on the base plane is transformed onto the view plane. Depending on the size and position of the view plane, some voxels may not contribute to the final image, which means that in the second phase rays may not need to be cast to cover the entire volume, and hence the processing time can be reduced further.

(a) 6-connected. (b) 26-connected.

Figure 2.3: Discrete ray representation.

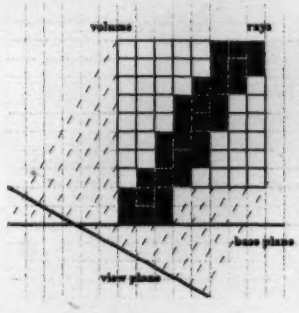


Figure 2.4: Discrete ray tracing algorithm.

2.2.3 Domain Volume Rendering

Direct volume rendering can also be performed in the frequency domain using the Fourier projection-slice theorem [63]. Once the volume is transformed from the spatial domain to the frequency domain, a 2D slice, which is centered at the volume origin and is parallel to the view plane, is extracted from the 3D spectrum. This 2D slice is then transformed back to the spatial domain to obtain the final image. The overall algorithmic complexity is $O(n^2 \log n)$ for a volume with size

n^3, in contrast to $O(n^3)$ for many spatial methods. One problem with frequency domain volume rendering is that it does not solve occlusion, which is an important cue to distinguish shapes and spatial positions of different objects. Totsuka and Levoy [102] solve this problem in an alternative way, where depth cueing and directional shading are employed. Depth cueing is implemented by frequency domain differentiation and directional shading is implemented by frequency domain multiplication. Entezari *et al.* [27] improve the illumination (an alternative to help determining 3D shapes) and achieve interactive rendering for constant diffusive light sources. Westenberg and Roerdink [110] incorporate wavelet decomposition into the Fourier volume rendering, which provides progressive refinement of the rendered images.

2.2.4 Hardware-Accelerated Volume Rendering

Texture mapping, initially introduced by Catmull [16], is a technique that adds details to 3D geometric models using photos. A texture can be considered as a mathematical function, and the domain of the function can be one, two, or three-dimensional and can be represented by either an array or by an analytical function [40]. Three dimensional (3D) texture mapping methods were implemented in software until the RealityEngine system [3] implemented this function in real-time. Hardware-accelerated texture mapping makes it possible for the computationally intensive interpolation operations used in direct volume rendering to be moved from the CPU to the graphics processing unit (GPU), which dramatically enhances the rendering speed. The normal procedure is illustrated at Figure 2.5. The volume is first loaded into the texture memory of the GPU, and sampled using trilinear interpolation to produce a set of equally-spaced slices (proxy polygons) parallel to the view plane. Then the slices are mapped with textures and blended from back to front to form the final image.

Cabral *et al.* [14] treat volume rendering as a generalized Radon transform and implement the algorithm on the RealityEngine hardware. In the 3D perspective case, a Radon transform can be defined for each ray as:

$$p(s,t) = \frac{d}{\sqrt{s^2+t^2+d^2}} \int_0^{l_{max}} \frac{c(l) f(x(l), y(l), z(l))}{r^2} dl \qquad (2.6)$$

where $f(x,y,z)$ denotes the volume space; $p(s,t)$ denotes the line integral projection of $f(x,y,z)$ along the ray $[x(l), y(l), z(l)]$, *i.e.*, the image space; $\frac{d}{\sqrt{s^2+t^2+d^2}}$ is

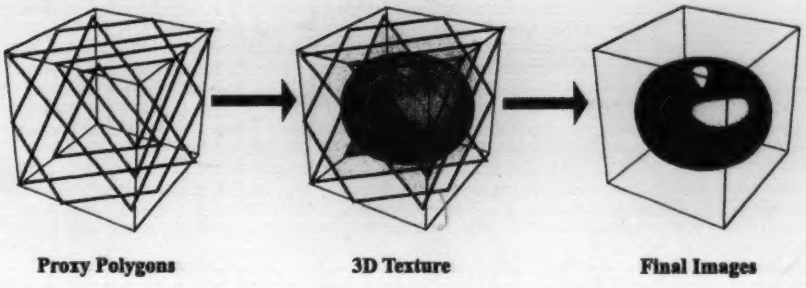


Figure 2.5: Volume rendering using 3D texture mapping.

the path length difference for off-center pixels; $c(l)$ is the blending weighting term; and $\frac{1}{r^2}$ is the perspective weighting term. A discrete form of this equation, which accumulates samples along the ray direction, is utilized for the implementation.

The algorithm proposed by Cabral *et al.* [14] does not take shading into account. Later, Van Gelder and Kim [106] incorporate shading into the hardware accelerator pipeline. The color of a texel (*i.e.*, a texture element) consists of an ambient component and a reflecting component. To calculate the ambient component, the volume is first divided into slabs as in image-aligned splatting [68]. Then the color intensity and opacity are evaluated slab by slab using the differential intensity and opacity equations proposed by Wilhelms and Van Gelder [113]. The reflecting component is calculated based on the cell-diagonal data shift s, a quantity derived from the voxel's quantized gradient vector and the inter-voxel spacing. According to the value of s, a voxel is assigned a probability of being on the boundary between different materials. If a voxel is on the boundary, it is assumed to be reflecting and its response to directional lighting is computed using the probability. For every possible combination of the quantized gradient and material, the color is calculated and stored in a lookup table to accelerate the rendering. However, this lookup table needs to be recomputed when rendering factors, such as light direction, are changed.

In Van Gelder and Kim's architecture [106], not only does the shading need to be recalculated when rendering factors are changed, but also the application of non-linear color transfer functions is impossible since the colors are interpolated using the hardware. Dachille *et al.* [21] improve this architecture in a way that the shading and compositing are moved to the main memory and CPU, and the hardware only

deals with ray sampling and perspective scaling (if perspective projection is used). Space-leaping is performed (in software) before Phong shading and front-to-back compositing take place to efficiently skip transparent regions. Unlike conventional texture hardware sampling methods that produce texture-mapped polygons parallel to the view plane, this method generates a set of polygons perpendicular to the view plane. The advantage of such oriented polygons is that the compositing can be performed one image scanline at a time and one polygon contributes to one scanline, like the shear-warp method [56], therefore, early ray termination can be employed.

Aside from object-order methods, image-order methods can also be implemented using 3D texture interpolation hardware. Guan and Lipes [35] introduce an innovative parallelization scheme for hardware-accelerated image-order volume rendering, in which a computation node operates on a subvolume and rays are sampled in the volume by parallel plane cutting. However, it only supports simple ray functions, such as maximum intensity projection.

To achieve higher rendering speed and quality using ray casting, specialized hardwares are designed. Cube-4 proposed by Pfister and Kaufman [75] is a parallel architecture that performs slice-parallel ray casting and is able to handle very high-resolution data sets (*e.g.*, 1024^3). Based on the Cube-4 architecture, Pfister *et al.* [74] develop VolumePro, the first single-chip real-time volume rendering system for consumer PCs, which performs orthographic projections at very high frame rate. The shear-warp factorization algorithm described in [56] is utilized, but trilinear interpolation is performed instead of bilinear interpolation. In addition, rays can be casted from sub-pixel locations. Therefore, view-dependent artifacts can be eliminated when the base plane changes. Several advanced features, such as supersampling and cropping, are also incorporated into VolumePro's architecture.

With the development of commodity graphics hardware, GPU-based ray casting can be implemented without specialized hardware. Krüger and Westermann [54] integrate early ray termination and empty space skipping into GPU-based ray casting. Before ray traversal, the entry points and ray directions are computed and stored in two separate 2D textures. The entry points are computed as the 3D texture coordinates of the intersection points between the volume's three front faces and the rays. The generated 2D texture has the same resolution as the current viewport, which guarantees one ray per pixel. For each ray entry point, the corresponding

exit point on the volume's three back faces is calculated. Then, each ray direction, *i.e.*, the difference between the exit and entry points, is stored as an RGBA pixel in the second 2D texture. During ray traversals, M equi-spaced samples are taken along each ray that is determined by the entry point and the direction stored in the two 2D textures. Between each ray traversal, a z-test is performed for early ray termination. Empty space skipping is applied via a min-max octree structure together with a 2D texture that stores the transparency information for every scalar range bounded by a different min-max pair. For the data sets with transfer functions that map a large amount of scalar values to low opacity, this method has superior performance to GPU-based object-order approaches.

Based on the method proposed by Krüger and Westermann [54], Scharsach [86] introduces some improvements for rendering pre-classified volumes. Empty space skipping is applied via a data-dependent bounding geometry instead of a min-max octree. Taking advantage of the vertex shader, a tighter bounding geometry other than a simple bounding box is utilized to exclude empty regions and regions of no interest while rendering. As for iso-surface rendering, a hitpoint refinement algorithm is proposed. Every time an iso-surface point is found with sample step size d along a ray, the point $\frac{1}{2}d$ back along the ray is checked to see whether it is also on the iso-surface. If it is, another point $\frac{1}{2}(\frac{1}{2}d)$ back along the ray is checked. By taking six such bisection steps, a volume can be sampled at 4 to 5 times the speed of the ordinary equi-spaced sampling. The resulting image is produced with uncompromised quality. In this algorithm, the whole volume needs to be loaded into the texture memory for ray casting, making rendering of large data sets impossible because of limited texture memory on current hardwares. Scharsach deals with this problem by only caching the regions of interest via a 2-way blocking scheme. Although this is not a radical solution, it makes possible the rendering of sparse large data sets with ray casting.

As for the frequency domain volume rendering, Viola and Kanitsar [107] move the rendering stage to the GPU and achieve a speedup factor of 17 compared to the CPU-based approach. After a pre-processing step on the CPU where the volume is transformed into frequency domain, slicing and interpolation, as well as the inverse fast Fourier transform, are all performed on the GPU. For 4D data, Binotto *et al.* [7] use a fragment-shader compression approach to achieve real-time volume rendering. Based on hierarchical vector quantization, Schneider and West-

ermann [89] propose a compression scheme for rendering both 3D and 4D volumetric data sets.

2.2.5 Critical Review of Volume Rendering Algorithms

In the previous sections, a review of most important direct volume rendering algorithms are presented. Different algorithms can be classified into three main categories, *i.e.*, object-order, image-order and domain methods. Hardware acceleration can be applied to all algorithms in the three categories. Some hybrid methods [36] [39] [67] are proposed by researchers in recent years, but their fundamental operations still fall into one of the three main categories. Meißner *et al.* [66] give a detailed comparison between the four most popular volume rendering techniques: ray casting, splatting, shear-warp and 3D texture hardware-based methods. Their experiments demonstrate that ray casting and splatting generate high-quality images at the cost of rendering speed, whereas shear-warp and 3D texture mapping hardware are able to maintain an interactive frame rate at the expense of image quality. When utilizing splatting for volume rendering, it is difficult to determine the parameters such as the type of kernel, the radius of the kernel, and the resolution of the footprint table to achieve an optimal appearance of the final image as described in [95]. In shear-warp, the memory cost is high since three copies of the volume need to be maintained. Most of current graphics cards support 3D texture, therefore hardware-accelerated methods can be easily applied on these cards. The frequency domain methods perform fast rendering, but they are limited to orthographic projections and X-ray type rendering [27]. In addition, there are two intrinsic problems with the frequency domain methods: high interpolation cost and high memory cost [102]. Hence, ray casting is a better choice for high-quality volume rendering, while hardware-accelerated approach is a better choice for high-speed volume rendering.

2.3 Virtual Environments

Bishop and Fuchs [8] define *virtual environments* (VE) to be real-time interactive graphics with three-dimensional models, when combined with a display technology that gives the user a sense of immersion in the model world and direct manipulation. A synonymous term for *virtual environments* is *virtual reality* (VR). According

to Burdea and Coiffet [13], *virtual reality* is a high-end user-computer interface that involves real-time simulation and interactions through multiple sensorial channels, *i.e.*, visual, auditory, tactile, smell and taste. Research on VE dates back to 1965 when Sutherland [98] proposed the ultimate display system. Since then, new techniques and systems have been continuously developed, and VR has been adopted in more and more applications, such as medicine [96], geoscience [32] and education [47]. As described in [10], to achieve immersive effects from VR, four crucial technologies are necessary: visual displays, graphics rendering system, tracking system, and database construction and maintenance system. Based on their display technology, VR systems can be divided into three categories: projection-based, monitor-based and HMD-based [76]. This section focuses on the major developments that one can find in the scientific literature on the visual displays used in VR, especially the ones found for medical applications.

2.3.1 Projection-Based VR Systems

Projection-based VR systems use rear-projection and/or front-projection screens to create stereopsis for the users. Typically the user needs to wear shutter or polarized glasses to see the stereo image. The CAVE (CAVE Automatic Virtual Environment) [18] [19], invented at the Electronic Visualization Laboratory at the University of Illinois at Chicago, is one of the most popular VR systems for scientific visualization. The CAVE is composed of a cubic space created by surrounding screens. Appropriate stereoscopic images are displayed on these screens and merged together to form a 3D virtual world. A small group of users (normally 5 to 10) can be in the CAVE simultaneously with one user serving as the guide, whose head movement is tracked to generate correct projecting images for the current view. A wand equipped with 3D tracker is usually used for various interactions.

The CAVE produces a large field of view (FOV) and even panoramic view. Nevertheless, due to its size, the CAVE cannot be deployed in offices. In contrast, at the expense of immersion, the Workbench [2] [55] provides a portable VE that is ideal for applications such as architecture and virtual prototyping. In the Workbench configuration, a rear-projection screen is laid horizontally in front of the user. The viewing parameters can be adjusted to set the virtual objects above or below the screen. A similar system called ImmersaDesk is developed by Czernuszenko *et al.* [20]. The difference is that in the ImmersaDesk setup the screen is placed at a

45-degree angle so that the user can look down as well as forward.

All the previous systems use active stereo where the images for each eye are projected onto the screen separately where the left and right shutters on the glasses become transparent and opaque alternately to let each eye see only the corresponding images. Unlike active stereo systems, passive polarized stereo systems achieve the stereo effect by using dual projectors with polarizing filters placed in front of each lens. Pape *et al.* [71] develop a low-cost dual-projector VR system based on LCD (liquid crystal display) projectors with polarized stereo glasses. A high-resolution stereo tiled display named ICWall is developed by van der Schaaf *et al.* [105] using a similar concept. Eight tiles are employed with one dual-projector for each tile. The projectors are automatically calibrated using geometric, photometric and viewpoint calibration to insure the left and right eye images are precisely aligned. This system is capable of generating a stereo image of roughly 2x4096x1524 pixels.

With the emergence of autostereoscopic displays, users do not need to wear special glasses but can see the stereo images with their naked eyes. Physically realizable autostereoscopic displays can be classified into three broad categories: re-imaging displays, volumetric displays, and parallax displays [37]. Re-imaging displays use optical effects to generate stereo illusion. The Cambridge display [24] uses red, green and blue CRT projectors to project a 3D scene onto a 50-inch concave spherical mirror. Different views of the scene are illuminated in turn. One common design of volumetric displays (*e.g.*, the DepthCube [97]) is to fill the space with display media and show a proper image on each layer. The Perspecta display produced by Actuality Systems Inc. [28] is 360-degree viewable by projecting hundreds of slices onto a fast rotating screen. Parallax displays emit lights with various intensities in different directions. When the user is at some special location, he or she can see the stereo image. More on parallax displays will be discussed under monitor-based VR systems.

2.3.2 Monitor-Based VR Systems

Normally, a monitor-based VR system cannot provide the immersion level that a projection-based system can, since the virtual scene is mixed with the workplace. Ware *et al.* [108] introduce the Fish Tank VR where a stereo image of a 3D scene is displayed on a monitor based on the user's head position, and with a pair of

shutter glasses the user can get a stereoscopic view.

Autostereoscopic displays mentioned in the previous section can also be monitor-based, but most of them are parallax displays. The DTI 3D display developed by Dimension Technologies Inc. [23] takes advantage of the screen parallax, the signed distance measured on the screen between two corresponding image points [109], to generate stereopsis. The left and right images are interlaced and then illuminated simultaneously by a special illumination plate to produce screen parallax. With a similar concept, Holografika Kft. devise the HoloVizio system [5]. Unlike the DTI 3D display, which requires the user to stay at specific positions, the HoloVizio display provides a much larger FOV, about 50 degree. Sandi *et al.* [85] introduce a large curved autostereoscopic display, Varrier display, by tiling 35 display panels together. The large FOV plus sub-pixel resolution gives the user a strong feeling of immersion.

2.3.3 HMD-Based VR Systems

Head-mounted displays (HMDs) are among the earliest display interfaces used in VR. The fundamental principle is that a perspective 2D image is shown for each eye, and the images are adjusted according to the head's movement, which gives the user an illusion that he/she is present in a 3D virtual world. The first HMD that uses CRTs was invented by Sutherland [99]. Over the decades, the resolution and color quality of HMDs increase dramatically, while the cost decreases due to the use of LCD [73] [101]. The resolution is not the only factor that affects the immersion provided by HMDs, and the other factor is the FOV (Field of View). The stronger a feeling of immersion is required, the larger a FOV is needed. Using hyperboloidal and ellipsoidal mirrors, Nagahara *et al.* [69] devise a wide FOV HMD, which provides a 180-degree horizontal view and a 60-degree vertical view including a 60-degree stereoscopic view at a resolution of about 7 pixels/degree. To attain a higher resolution while maintaining a relatively large FOV, a trade-off between the two factors is required since a limited number of pixels need to be spread on the microdisplay(s) across the FOV. Yoshida *et al.* [121] resolve the issue by tracing the eyes' gaze point and providing high resolution for that region of interest (ROI). The horizontal FOV of this High-Resolution Insert HMD is 50 degree at a resolution of 8 pixels/degree, and its ROI is 12.5 degree at a resolution of 32 pixels/degree. A recent technological breakthrough by Sensics Inc. [94] achieves a 180-degree hor-

izontal FOV at a resolution of 20 pixels/degree, using an array of microdisplays combined seamlessly into one large image that warps round the eyes. A detailed discussion on the current techniques and applications of HMDs can be found in a survey by Cakmakci and Rolland [15].

2.4 Visualization in Virtual Environments

In the field of scientific visualization, virtual reality offers a natural interface between human and computer that simplifies complicated data manipulations [81]. Due to such significance of VR technology, various visualization systems have been proposed for virtual environments over the decades. Some are designed for general-purpose visualization [42], while others are devised for particular applications, such as medical analysis [30] and natural phenomena simulation [12].

2.4.1 VR Systems for General-Purpose Visualization

Rajlich [77] develops a visualization framework that is able to work both in a CAVE and on a desktop. The design is based on the Visualization Toolkit (VTK) [91] and IRIS Performer [83]. The VTK pipeline and the Performer scenegraph are connected by a translator, vtkActorToPF. This combination enables the usage of VTK's visualization algorithms and Performer's multi-channel rendering capability that supports simultaneous rendering for multiple screens. Three major components are involved: VisGen, Application and Interface. The VisGen component is built on VTK to generate geometries, which are then passed to the Application component that is in charge of the state of the system. The interface component is the graphical user interface (GUI). The separation of different functional modules facilitates the transfer from the CAVE tool to the desktop tool. The user is presented with stereo images in the CAVE environment, but only 2D images on the desktop. Surface rendering is employed and the user can view different iso-surfaces by changing the iso-value parameters.

Schulze *et al.* [93] introduce a volume rendering system for visualizing scalar volumetric data in the CUBE, a CAVE-like virtual environment. The system is built on COVISE [78], a general visualization framework, and COVER [78], a VR interface library integrated with COVISE that supports basic rendering and tracking in VE. Hardware-accelerated direct volume rendering is accomplished by a plug-in

module for COVER, called VIRVO. Two rendering modes are incorporated: the adaptive mode with coarse image quality to maintain a high rendering speed and the high-quality mode where the user can investigate the data in detail. The user can use a transfer function editor to interactively manipulate the color and opacity mapping functions. Once a specific scalar value is selected, virtual rotary knobs are used to adjust the mapping from the scalar value to color and opacity. A probe mode is developed to help the user focus on the ROI, where a user-controlled cubic subset of the volume is rendered. Visualization of time-dependent volumetric data sets is also supported. However, since the entire time series is loaded into the texture memory before rendering, only small 4D data sets can be rendered at interactive speed.

While the previous systems are specialized for visualizing static volumetric data with little or no consideration for time-varing data, Jaswal [42] develops a distributed system, CAVEvis, to visualize time-dependent large scalar or vector field data in the CAVE. The visualization task is distributed among a number of modules running on different computers, which work asynchronously to maintain high rendering speed. Iso-surfaces are rendered for scalar field data and flows of particles are rendered for vector field data. The whole data set is randomly accessible to the user in both the spatial domain and the temporal domain.

2.4.2 VR Systems for Medical Visualization

Visualization in VEs has been proven useful in medicine for applications ranging from medical education, surgical training and planning, to the enhancement of minimally invasive surgeries [82].

Forsberg *et al.* [30] develop an immersive system for visualizing simulated blood flow through an artery model in the CAVE, aiming at understanding the factors that may cause the failure of coronary artery grafts. The flow is modeled numerically and simulated via particle techniques. The artery is rendered as a triangulated mesh and the shear stress, an essential quantity to determine the flow behavior, is encoded into the arterial wall color. An important feature of this system is its gestural and voice interaction techniques. The user can use different tools to examine the flow behavior, such as throwing a virtual particle into a particular region of the flow. The tools are selected from a virtual toolkit using voice commands and/or gestures when the user looks down.

While the previous system is for visualizing simulated medical data, many systems are designed to visualize real medical data acquired from imaging sensors such as CT and MRI. Zhang *et al.* [123] devise a system for visualizing diffusion tensor magnetic resonance imaging (DT-MRI) data sets of brains in a CAVE environment. Its potential applications are in the study of changes in white matter structures before and after gamma-knife capsulotomy, and pre-operative planning for brain tumor surgery. Hence, the major concern is the representation of neural structures and anatomical context in the brain. The fibrous structures in the regions of linear anisotropy and the planar structures in the regions of planar anisotropy are represented as streamtubes and streamsurfaces respectively. The ventricles are represented as iso-surfaces produced by the marching cubes algorithm [62]. All the generated geometric models are simplified beforehand to achieve a higher frame rate. A yellow line is drawn according to the position and direction of the wand to serve as a virtual pointer for indicating the region of interest. Traditional 2D image slices can also be displayed together with the geometric models to help the doctors, who are already familiar with the conventional analytical techniques, to identify the anatomical structures in a 3D virtual environment. However, due to a large amount of decimation required to obtain an interactive rendering speed, many details are lost in the final rendered models making it useless for routine radiological observations.

Lapeer *et al.* [57] introduce the ARView, a generic software framework for stereoscopic augmented reality microsurgery, which combines the surgical scene captured by a stereo pair of cameras in real time and the images pre-acquired from medical imaging modalities. For instance, a baby's skull and its brain are displayed together on an HMD. The surgical scene is surface rendered, while the pre-acquired medical images stored in the DICOM or raw format are volume rendered with the assistance of 2D or 3D texture hardware. The design of the ARView follows the factory design pattern [33], in which a high-level factory class deals with initiating the appropriate sub-class based on the computer system's software and hardware configurations. This kind of design enhances the reusability of the software framework. One drawback of the ARView lies in the rendering part, as it only produces gray-level images and does not support run-time modification to the transfer functions.

Kratz *et al.* [53] integrate high-quality perspective direct volume rendering into

the Studierstube [88], a virtual reality system developed in their laboratory. The rendering part is built on the hardware volume rendering framework introduced by Scharsach *et al.* [87]. Shaded iso-surface rendering and unshaded direct volume rendering are combined to enhance depth perception and visualization of boundaries. Volume rendered images are also combined with polygon rendered widgets by blending the geometry image on top of the volume. Several types of interaction are supported: navigation, clipping cube, lighting manipulation, slice viewer and fly-through. However, the transfer functions can only be designed off line and loaded at runtime. This inability to interactively manipulate the transfer functions impairs the effectiveness of analyzing the data as it relies on prior information to display the data set. This is contrary to the needs of practicing radiologists who want to have the minimal interpretation performed on the data set by the computer as those interpretation may hide subtle structures that are the important ones they are looking for.

Kniss *et al.* [52] apply hardware-accelerated volume rendering and VR techniques to visualizing multi-field medical data sets. Two dual-projector screens are placed side by side to provide 3D stereoscopic imagery, and a desktop PC is placed in front of one screen to provide 2D classification, *i.e.*, manipulation of the transfer functions. The 2D classification interface is built on the multi-dimensional transfer functions proposed in [51]. 3D classification interface is also supported, and like in [93], rotary knobs are used for adjusting the mapping from a scalar value to visual attributes. The visualization in VR is built on the same framework used in [93], *i.e.*, COVISE and COVER, and volume rendering is performed via the integration of Simian [50], a multi-field volume rendering tool. The system is limited to a fixed environment and cannot be extended to various display modalities that exist in real-world radiology department ranging from radiologist desktop to the operating room. In addition, the system cannot scale as data modalities get more complex and larger such as doppler MRI or temporal CT, desktop PCs cannot easily scale in bus bandwidth and memory.

2.5 Summary

Volumetric data rendering has become an important tool in various medical procedures as it allows the unbiased visualization of fine details of volumetric med-

ical data (CT, MRI, fMRI). However, due to a large amount of computation involved, the rendering time increases dramatically as the size of the data set grows, even when using GPU-based methods. In this thesis, we propose several acceleration techniques of volume rendering using general-purpose GPU. Some techniques enhance the rendering speed of software ray casting based on voxels' opacity information, while the other implementations improve traditional hardware-accelerated object-order volume rendering.

Some [30] [123] of the previous medical visualization systems support only surface rendering. Although they may be useful in a specific area, they are not suitable for a radiologist, who needs the fine details of the data set, to use in his or her daily workflow. Our system, MedVis, has integrated the advantages of many of the systems found in the literature, as MedVis supports enhanced GPU-based volume rendering, allowing for various real-time interaction modalities, and the ability of extensibility and portability.

Chapter 3

System Design and Implementation

3.1 Overview

The system, called *Medical Visualizer* (or *MedVis* for short), supports visualization and interaction with DICOM medical data sets in various virtual reality setups, such as parallax barrier autostereoscopic displays, dual-projector screens, and the CAVE. Multiple direct volume rendering techniques, *i.e.*, software-based ray casting and GPU-based object-order volume rendering, are integrated to meet different visualization requirements and to accommodate different hardware configurations. The whole system is designed in a modular fashion and consists of three major elements: the kernel module, the desktop interface module and the CAVE interface module. The kernel module deals with all VR-independent operations; the desktop interface module exports the kernel functions to the PC-based VR systems, like autostereoscopic displays; and the CAVE interface module extends the visualization pipeline into the immersive CAVE. Due to this modular design, MedVis can be easily extended to support other VR setups, such as the Workbench [2] or the GeoWall [43]. The implementation of MedVis is platform-independent. Although MedVis is currently running under Windows, its cross-platform nature allows it to run under Linux, IRIX etc. with minimal modifications.

3.2 Rendering Algorithms

As stated in Section 2.2, surface rendering and direct volume rendering can both be applied to visualize volumetric data. However, surface rendering depends on the assumption that the important structures in medical images can be segmented us-

ing simple algorithms such as the marching cubes. This is contrary to most medical practice as radiologists need to see the data set as it is, as simple minded computer interpretation may hide important structures [66] required to perform a diagnosis. On the contrary, direct volume rendering bypasses the intermediate geometric representation and directly renders the volumetric data set based on its scalar values. This allows a radiologist to visualize the fine details of medical data as he/she is the one who by law must make the final interpretations. Considering the advantages for radiology of direct volume rendering over standard surface rendering, this is why direct volume rendering (DVR) is employed in MedVis as the main rendering algorithm.

3.2.1 Direct Volume Rendering

As mentioned in Section 2.2.5, ray casting is a better choice for high-quality volume rendering, and hardware-accelerated approach is a better choice for high-speed volume rendering. In order to accommodate different system configurations, software-based ray casting and GPU-based object-order DVR are both incorporated into MedVis.

Software-Based Ray Casting Volume Rendering

Software-based ray casting approach casts one ray per screen pixel into the volume. Samples (trilinear interpolated) are taken along each ray and the color of each sample is accumulated from front to back to form the final color of the current pixel. This approach produces high-quality images, but due to the huge amount of calculation, the basic algorithm suffers from poor real-time performance. To accelerate software-based ray casting, several enhancements are proposed in this thesis.

As mentioned in Section 2.2.2, there are two common acceleration techniques for ray casting volume rendering: early ray termination and empty space skipping. Early ray termination exploits the fact that when a region becomes fully opaque or of high opacity, the space behind it can hardly be seen. Therefore, ray tracing stops at the first sample point where the remaining opacity is less than a user-specified threshold.

Empty space skipping is achieved via the use of a precomputed min-max octree structure. It can only be performed efficiently when classification is done before interpolation, *i.e.*, when the scalar values in the volume are converted to colors before

(a) Classification first. (b) Interpolation first.

Figure 3.1: The comparison of classification first and interpolation first.

the volume is resampled. However, this often produces coarser results than applying interpolation first. As compared in Figure 3.1[1], the volume-rendered images of a whole pelvic region data set using classification first (Figure 3.1(a)) are blurrier than those rendered using interpolation first (Figure 3.1(b)), and the blur is most obvious in the spinal area. If empty space skipping is applied with interpolation prior to classification, one additional table lookup operation is needed to determine whether there are non-empty voxels in the current region. Nevertheless, the major drawback with this empty space skipping technique lies in that every time the transfer functions change, the data structure that encodes the empty regions or the lookup table need to be updated. Instead, an intuitive empty space skipping technique is employed in MedVis: only the non-transparent sample points are involved in the color accumulation.

However, only with early ray termination, the rendering speed is still often far from satisfactory, even for medium-size data sets (*e.g.*, 256^3). Therefore, another acceleration technique is necessary. Jittered sample interval borrowed from the β-acceleration [22], is applied together with early ray termination. The basic idea is that the sample interval along each ray becomes larger as the pixel opacity accumulates. Unlike the β-acceleration, which depends on a pyramidal organization of the volumetric data, the jittered sample interval is applied directly to the data set in MedVis. We term the modified β-acceleration as β'-acceleration. Instead of going

[1] All of the medical data used in this thesis is provided by the Department of Radiology & Diagnostic Imaging, University of Alberta.

up one level in the pyramid whenever the remaining pixel opacity is less than a user-defined threshold after a new sample is taken, the sample interval is modified according to a function of the accumulated pixel opacity:

$$s = s \times (1.0 + \alpha \times f) \tag{3.1}$$

where s denotes the length of the sample interval; α denotes the accumulated opacity; and f is a predefined jittering factor. The initial value of s is set by the user. Normally, the smaller s is, the better image quality. For every sample point, the remaining opacity γ is compared against a user-specified threshold. If γ is less than the threshold, the current sample interval is adjusted according to Equation 3.1.

To further enhance the performance of software-based ray casting during interaction, the sample interval is automatically enlarged to maintain a high rendering speed, and once interaction stops, the sample interval is set back to normal. When multiple processors are available, the viewport is divided into several regions and each processor handles its own region.

Algorithm 3.1 MedVis ray casting algorithm.

```
    Break current viewport into N regions of equal size
    Initialize early ray termination threshold Γ
    Initialize jittering start threshold Γ'
    Initialize jittering factor f
 5: Initialize sample interval s
    for each region do
      for every pixel in the current region do
        Compute ray entry point, direction, maximum tracing distance D
        while the traced distance d < D and γ < Γ do
10:       Interpolate at current sample point
          Get opacity value α according to opacity mapping function
          if α ≠ 0 then
            Compute pixel color according to color mapping function
            γ ← γ × (1.0 − α)
15:       end if
          if γ < Γ' then
            s ← s × (1.0 + (1 − γ) × f)
          end if
          d ← d + s
20:       Compute next sample position
        end while
      end for
    end for
```

The whole process is executed in CPU using main memory. The enhanced al-

gorithm is illustrated at Algorithm 3.1. Not only is this parallel software approach suitable for computers with low-end graphics cards, but since parallel ray tracing is employed, it is also suitable for multi-processor computers or clusters.

GPU-Based Object-Order Volume Rendering

GPU-based object-order volume rendering has several advantages over GPU-based ray casting. First, perspective projections can be more easily implemented with object-order volume rendering, since only a proper scaling factor needs to be assigned to each slice based on several viewing parameters while in ray casting the direction of each ray needs to be determined individually. Second, as pointed out in [86], GPU-based ray casting has the limitation that it can only render volumes that fit in the texture memory. Since ray tracing needs to randomly access the whole volume, it is impossible to break the volume into subvolumes and load each subvolume only once per frame. Finally, most of the speedup from GPU-based ray casting comes from empty space skipping, and ray casting with only early ray termination shows performance comparable to object-order volume rendering if implemented in the GPU (see comparison in [54]). In addition, the object-order volume rendering has a more regular processing structure, in which the volume is processed slice by slice, therefore it is more suitable to be implemented in hardware. Hence, the GPU-based object-order volume rendering is employed in MedVis.

The rendering algorithm is similar to the one proposed by Rezk-Salama and Kolb [80], which balances the workload between the vertex shader and the fragment shader. Most of previous implementations generate the proxy polygons in the CPU and use the fragment shader for trilinear interpolation and texture mapping. Little work has been done to exploit the vertex shader in the hardware-accelerated volume rendering pipeline. Based on the observation of different box-plane intersection cases, the generation of proxy polygons can be moved from the CPU to the GPU. The intersection between a proxy plane and the bounding box of the volume has five different cases, ranging from 3 intersection points to 6, as illustrated at Figure 3.2. Let $\vec{n} \cdot (x, y, z) = d$ represent a plane, where $\vec{n}$ is the normalized plane normal and d is the signed distance between the origin and the plane, and let $V_i + \lambda \vec{e}_{i,j}$ represent the edge $E_{i,j}$ from vertex V_i to V_j, where $\vec{e}_{ij} = V_j - V_i$, then the

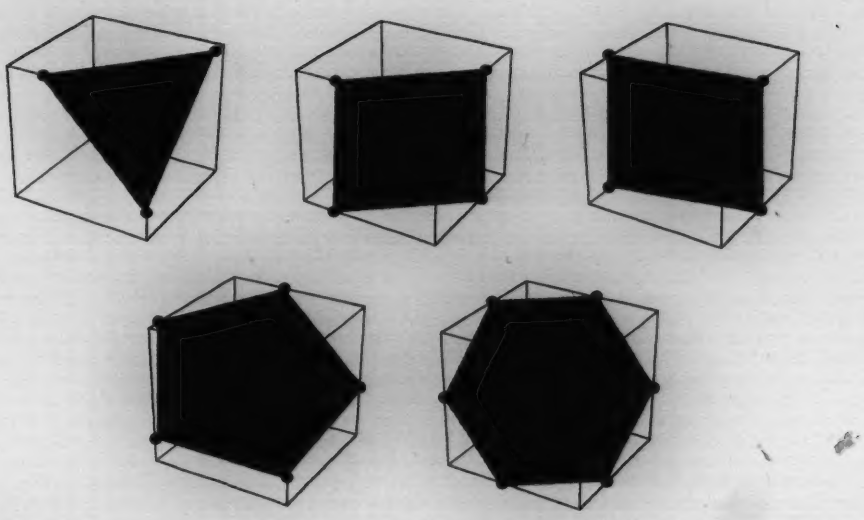


Figure 3.2: The intersection cases between a proxy plane and the volume bounding box.

intersection between the plane and the edge can be computed by

$$\lambda_{i,j} = \begin{cases} \frac{d-\vec{n}\cdot\mathbf{V}_i}{\vec{n}\cdot\vec{e}_{i,j}}, & \vec{n}\cdot\vec{e}_{i,j} \neq 0; \\ -1, & \text{otherwise.} \end{cases} \tag{3.2}$$

If $\lambda_{i,j} \in [0, 1]$, there is a valid intersection; otherwise, no intersection.

The edges of the volume bounding box are checked following a specific order, so that the intersection points can be obtained as a sequence that forms a valid polygon. If V_0 is the front vertex (the one closest to the viewpoint) and V_7 is the back vertex (the one farthest from the viewpoint), then the edges are divided into six groups, as shown in Figure 3.3 marked with different colors. For a given plane parallel to the viewport that does intersect with the bounding box, there is exactly one intersection point for each of the three groups (red, blue and green), and at most one intersection point for each of the other three groups (yellow, cyan and purple). The six intersection points P_0 to P_5 are computed in the way described in Table 3.1. For the other seven pairs of front and back vertices, the only extra computation is to map each vertex to the corresponding vertex in this case, which can be implemented as a simple table lookup.

In Rezk-Salama and Kolb's method [80], the coordinates of a sample point in

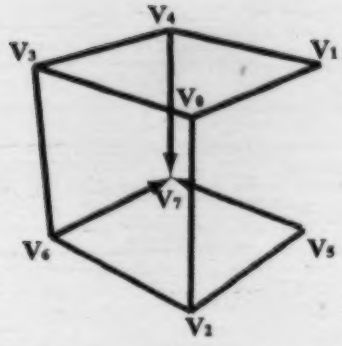


Figure 3.3: The traversal of the bounding box edges.

Table 3.1: The computation of the intersection points.

Point	Checked Edges	Intersection Position
P_0	The red edges: $E_{0,1}$, $E_{1,4}$ and $E_{4,7}$	$\lambda_{i,j}$, where $(i,j) \in \{(0,1),(1,4),(4,7)\} \wedge \lambda_{i,j} \in [0,1]$
P_1	The yellow edge: $E_{1,5}$	$\begin{cases} \lambda_{1,5}, & \lambda_{1,5} \in [0,1]; \\ P_0, & \text{otherwise.} \end{cases}$
P_2	The green edges: $E_{0,2}$, $E_{2,5}$ and $E_{5,7}$	$\lambda_{i,j}$, where $(i,j) \in \{(0,2),(2,5),(5,7)\} \wedge \lambda_{i,j} \in [0,1]$
P_3	The cyan edge: $E_{2,6}$	$\begin{cases} \lambda_{2,6}, & \lambda_{2,6} \in [0,1]; \\ P_2, & \text{otherwise.} \end{cases}$
P_4	The blue edges: $E_{0,3}$, $E_{3,6}$ and $E_{6,7}$	$\lambda_{i,j}$, where $(i,j) \in \{(0,3),(3,6),(6,7)\} \wedge \lambda_{i,j} \in [0,1]$
P_5	The purple edge: $E_{3,4}$	$\begin{cases} \lambda_{3,4}, & \lambda_{3,4} \in [0,1]; \\ P_4, & \text{otherwise.} \end{cases}$

the world coordinate system are required to be the same as the coordinates of the corresponding sample point in the texture coordinate system. However, this is not true for most cases, where the size of the volume in the data coordinate system or the world coordinate system is not the same as that in the texture coordinate system. In MedVis, the box-plane intersection test is carried out in the data coordinate system. Since typically the texture coordinates need to be normalized to a range between $[0,1]$, a conversion of the valid intersection points' coordinates is required. If the point P_k intersects the edge $E_{i,j}$ at the proportional position $\lambda_{i,j}$, then each

coordinate of the resulting texture-space intersection point P'_k is obtained by:

$$P'_k.p = \begin{cases} \frac{V_i.p - \min(B_p)}{\max(B_p)}, & \vec{e}_{i,j}.p = 0; \\ \lambda_{i,j}, & \vec{e}_{i,j}.p > 0; \\ 1 - \lambda_{i,j}, & \vec{e}_{i,j}.p < 0. \end{cases} \tag{3.3}$$

where p denotes either x, y or z; B denotes the volume bounding box; and B_p denotes the length of B in the p-direction. The coordinates of P'_k are then scaled and translated in order to sample near the center of the cubic region formed by eight adjacent voxels in the texture memory.

One enhancement with respect to the rendering speed is also incorporated: the sample interval is adjusted based on the size of the volume in the world coordinate system and the distance from the viewpoint to the volume. This idea of adaptive sample interval is like level-of-detail (LOD) in mesh simplification. The sample interval is calculated by:

$$s = S \times F^{\frac{\max(d)}{\max(B_x, B_y, B_z)}} \tag{3.4}$$

where S denotes the constant initial sample interval; $F \geqslant 1$ denotes the predefined interval scale factor; B_x, B_y, and B_z denote the length of the volume bounding box B in the x, y, and z-direction respectively; and $\max(d)$ denotes the distance between the farthest vertex of B and the view plane.

Now that the proxy polygons are generated, one can then perform texture mapping. The fragment shader performs two texture lookups per fragment to attach the textures onto the proxy polygons. The first texture lookup gets the scalar value associated with the sample point from a 3D texture that holds the volumetric data. The hardware does the trilinear interpolation automatically for every sample. The second texture lookup uses the scalar value to get the corresponding color from a 2D texture that encodes the transfer function. Then, the textured polygons are written into the framebuffer from back to front to produce the final image. The dataflow between the CPU and the GPU is illustrated in Figure 3.4.

The vertex program and the fragment program are both written in Cg, a high-level shading language developed by NVIDIA. To exploit the most powerful profile supported by a graphics card, the shader programs are compiled at runtime instead of at compile time. Rezk-Salama and Kolb's method requires at least *OpenGL NV_vertex_program 3.0 profile* to compile. However, to accommodate graphics cards with different vertex processing capabilities, the amount of work assigned to the vertex shader should vary from card to card. The more capable the programmable

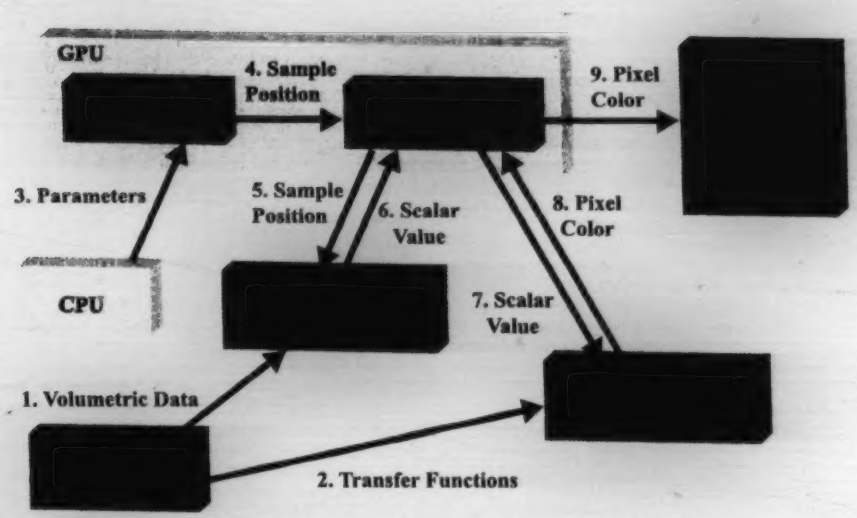


Figure 3.4: The dataflow between the CPU and the GPU.

graphics hardware is, the larger the amount of processing are moved from the CPU to the vertex shader. Currently, the vertex program has variations for all the OpenGL vertex program profiles supported by the Cg compiler. The fragment program only requires *OpenGL ARB fragment program profile* or *OpenGL NV_texture_shader and NV_register_combiners profile* to compile, which are supported by all graphics cards that have fragment shader programmability. Therefore, theoretically the proposed GPU-based volume rendering program can be executed on any computer with a programmable graphics card.

Gradient Opacity Mapping

Gradient opacity mapping can help amplify the boundaries between different materials, which can be used together with scalar color mapping in the classification of the volume. The gradient g at every voxel is evaluated using finite difference method. If the voxel does not lie on the volume boundary, its gradient is calculated

by the central difference:

$$
\begin{aligned}
g_x &= \frac{f(x+\frac{1}{2}\Delta x, y, z) - f(x-\frac{1}{2}\Delta x, y, z)}{\Delta x} \\
g_y &= \frac{f(x, y+\frac{1}{2}\Delta y, z) - f(x, y-\frac{1}{2}\Delta y, z)}{\Delta y} \\
g_z &= \frac{f(x, y, z+\frac{1}{2}\Delta z) - f(x, y, z-\frac{1}{2}\Delta z)}{\Delta z}
\end{aligned}
\tag{3.5}
$$

where $f(x, y, z)$ denotes a function that returns the scalar value at volume location (x, y, z). If the voxel is on the volume boundary, then at least one of its three gradient components needs to be calculated by the forward or backward difference. For instance, if the voxel is located at the bottom-left vertex of the back face of the volume, all of its three gradient components are calculated using the forward difference.

$$
\begin{aligned}
g_x &= \frac{f(x+\Delta x, y, z) - f(x, y, z)}{\Delta x} \\
g_y &= \frac{f(x, y+\Delta y, z) - f(x, y, z)}{\Delta y} \\
g_z &= \frac{f(x, y, z+\Delta z) - f(x, y, z)}{\Delta z}
\end{aligned}
\tag{3.6}
$$

In another extreme case when the voxel is located at the top-right vertex of the front face of the volume, all of its three gradient components are calculated using the backward difference:

$$
\begin{aligned}
g_x &= \frac{f(x, y, z) - f(x-\Delta x, y, z)}{\Delta x} \\
g_y &= \frac{f(x, y, z) - f(x, y-\Delta y, z)}{\Delta y} \\
g_z &= \frac{f(x, y, z) - f(x, y, z-\Delta z)}{\Delta z}
\end{aligned}
\tag{3.7}
$$

After the three components of a gradient $\vec{g}$ are evaluated, the magnitude $\|\vec{g}\|$ is mapped to an opacity value α'. Now, the final opacity of a voxel is $\alpha' \times \alpha$, where α is the opacity value obtained from the scalar opacity mapping. When the gradient opacity mapping is incorporated, the transfer function moves from 1D to 2D domain. When integrated with the GPU-based volume rendering, like the 1D case, the 2D transfer function is also encoded in a 2D texture. The s-coordinate corresponds to the scalar value and the t-coordinate corresponds to the gradient magnitude. Each pixel of the 2D texture stores the quantized RGBA color associated with the corresponding scalar-gradient pair.

3.2.2 Stereo Rendering

Stereo rendering takes advantage of the *binocular parallax*, the horizontal difference in the position of one object seen by each eye. The brain merges the two slightly different 2D images. Therefore, if the rendering system generates the two images based on each eye's viewing positions and delivers each image to the corresponding eye, the user can experience stereopsis. As pointed out by Hodges [41], there are two ways to present the left and right images: time parallel and time multiplexed. Time parallel methods display the two images simultaneously on a single display or two separate displays. In the one-display case, the stereopsis is generated either by interlacing the images inside the display (*e.g.*, the DTI Virtual Window) or by exploiting some external optical devices (*e.g.*, the dual-projector polarized passive stereo display). In the two-display case, *i.e.*, the HMD, the generation of stereo images is much easier. Since each small display is very close to both eyes, each eye can only see the images shown on the display right in front of it. When the two images are displayed separately, no extra work is needed for delivering the correct images for each eye. Time multiplexed methods display the two images alternately on a single display. With a pair of shutter glasses, the frequency of which is synchronized with the display, at every time point, the currently displayed image is always delivered to the corresponding eye. When the alternate images are displayed in sequence at 120 Hz or more, the user can see a flicker-free 3D image.

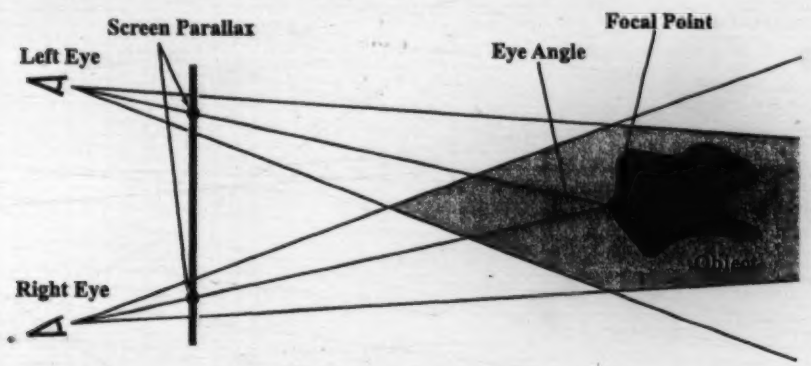


Figure 3.5: Stereo rendering.

For both methods, the most important task is to generate the correct left and right images based on the binocular parallax. When a virtual object is observed through a display (or screen), the binocular parallax can be considered as an effect

of the screen parallax. Figure 3.5 illustrates how the screen parallax occurs. The same point of the object is projected for each eye's respective viewing direction, which results in two different footprints on the screen (or view plane in the world coordinate system). To find the exact two positions that a point of the object is projected to, the eye positions, the eye separation distance and the eye focal point need to be obtained first. Based on this information, the left and right screen footprints of an object can be calculated. The average eye separation distance is 6.35 cm. Alternatively, the eye angle can also be used to control the amount of parallax, since the relative position of the virtual object to the view plane(s) can be easily calculated. In MedVis, we use the eye angle to control the amount of the screen parallax.

In practice, two virtual cameras are used to simulate the eyes in the world coordinate system. For such time-parallel systems as the HMD, which uses two displays, when each camera is positioned along the corresponding eye's viewing direction, a simple perspective projection to each of the two view planes (yellow line segments in Figure 3.5) can generate the correct left and right images. For other time-parallel systems with one display and time-multiplexed systems, the left and right images share the same view plane (blue line segment in Figure 3.5). In this case, an additional oblique projection is performed to project the images from each camera's respective view plane to the shared view plane.

3.3 Hardware Setup

3.3.1 Desktop Version

In the desktop configuration MedVis runs on a consumer PC with an autostereoscopic display. The current configuration uses an 18-inch DTI autostereo display (on the left) to deliver glassless stereoscopic imagery and a normal display (on the right) to display the control interface, as shown in Figure 3.6.

As mentioned in Section 2.3.2, the DTI 3D display developed by Dimension Technologies takes advantage of screen parallax to generate stereopsis. A special illumination plate is placed behind an LCD screen, so that the strips of the left and right eye images are interlaced across the screen. Once the plate is illuminated, the space before the screen is divided into alternate left eye and right eye zones the so called sweet spot. When the user's left and right eyes are positioned in the left eye

Figure 3.6: MedVis desktop hardware setup.

and right eye zones respectively, the left eye can only see through the odd columns of the LCD screen and the right eye can only see through the even columns. Hence, each half of the pair of stereo images is delivered to the corresponding eye. As for interaction, a 2D mouse is used to manipulate the volume or the control panel. Since this version only requires to add one autostereoscopic display to a normal PC setup, it can easily fit on a radiologist's workplace.

3.3.2 Immersive CAVE Version

Our CAVE is a $10' \times 10' \times 8'$ cube with rear-projected front, right and left walls, as shown in Figure 3.7. The CAVE version runs on a cluster, where each cluster node handles one of the three walls.

A Microsoft joystick (Figure 3.8) or InterSense motion tracker is used for navigation or interaction. The current configuration employs an InterSense IS-900 (Figure 3.9), a 6-DOF (degree of freedom) inertial-acoustic motion tracking system. A head tracker is placed on the user's head or the shutter glasses, and a tracked wand with 4 buttons is used by the user as a controller.

Figure 3.7: MedVis CAVE hardware setup.

Figure 3.8: Microsoft force feedback joystick.

(a) Head tracker.

(b) Wand.

Figure 3.9: InterSense IS-900.

3.4 Interaction Modalities

MedVis supports several types of interaction that helps analyze the medical volumetric data, such as real-time manipulation of the transfer functions and transformation of the displayed volume. Although the basic concepts remain the same from the desktop interaction mode to the CAVE interaction mode, there is some difference between them.

3.4.1 Desktop Interaction

In the desktop interaction mode, the transfer functions can be adjusted through a 2D widget, and for all the transfer function changes, the volume-rendered images are updated correspondingly in realtime. The style of the widget resembles the one used in VolView. Figure 3.10 shows the control panel.

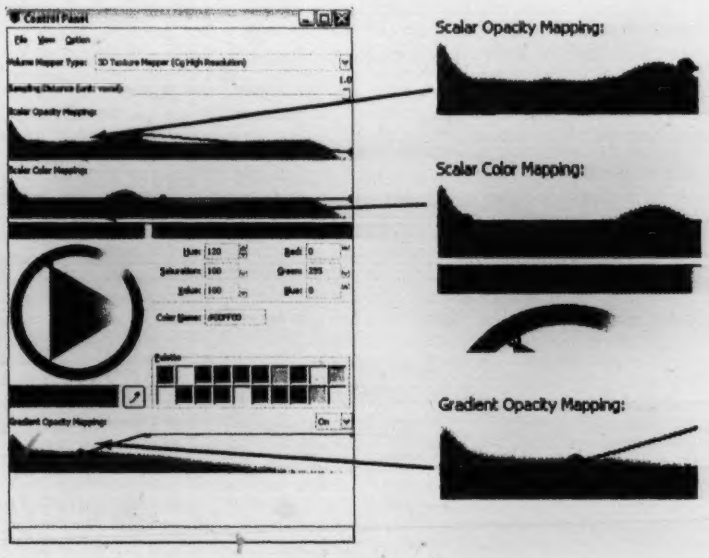


Figure 3.10: The desktop control panel.

The middle region of the control panel holds the transfer function editor. The top part of the widget controls the scalar opacity mapping. Its background shows the log-based histogram of the current scalar data. The foreground is the piecewise linear opacity transfer function represented as a combination of lines and spheres. The spheres are the control points that actually determine the shape of the transfer function. Once a control point is selected, it is highlighted by changing its color from blue to red and enlarging the radius. Once the control point loses focus, the color and size are back to normal. Except for the two control points at the ends, other control points can be added, deleted and moved using a mouse to change the scalar opacity mapping. The end control points can only be moved upward or downward and cannot be deleted for a transfer function to exist.

The middle part controls the scalar color mapping. The color representation is based on the Hue, Saturation and Value (HSV) model. Below the histogram,

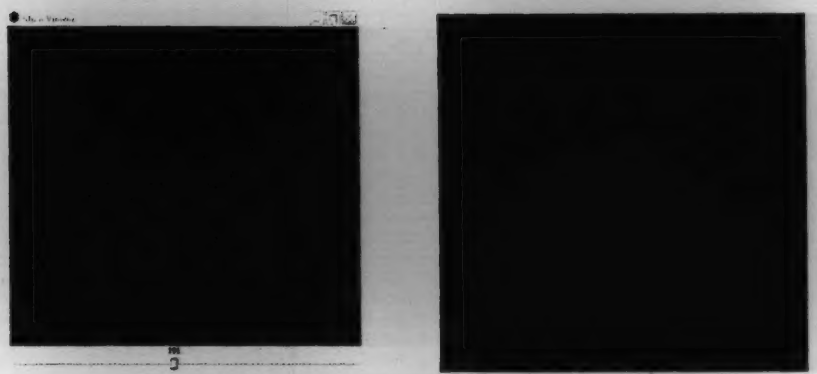
which is defined the same as in the scalar opacity mapping, is a color bar that displays the current color transfer function. Once a control point is selected, the palette right below the color bar is enabled and the user can change the color for the current selected scalar value. Like in the scalar opacity mapping, except the two control points on the ends, other control points can be added, deleted and moved horizontally using a mouse to change the scalar color mapping.

The bottom part controls the gradient opacity mapping. Its background is the log-based histogram of the gradients computed from the current scalar data. The control points can be moved by a mouse but cannot be deleted nor added. The use of the gradient opacity mapping is optional, which can be turned on/off at any time.

Four interaction styles for transforming the displayed volume (*e.g.*, rotation, panning, *etc.*) are supported: joystick camera, joystick actor, trackball camera and trackball actor. A mouse is used to simulate a joystick or trackball. The joystick styles perform transformation based on the position of the mouse, while the trackball styles perform transformation based on the magnitude of the mouse motion. The joystick-camera and trackball-camera styles normally guarantee a higher rendering speed than the other two, because they move the camera instead of transforming the volume. The volume can be re-sliced along the current viewing direction and the slices are displayed in a separate window. Figure 3.11(a) shows the slice viewer with slices generated from the pelvic volume displayed in Figure 3.11(b). Those interactions are defined in the InTml framework [29] to guaranty easy re-targeting to other VEs. InTml is an XML-based language developed at the University of Alberta for the description of complex VR application systems that allow to create formal model of interaction that is hardware-independent and component-based, allowing for easy re-targeting and code re-use.

3.4.2 CAVE Interaction

In the CAVE interaction mode, the transfer function editor is a 3D extension of the 2D widget used in the desktop interaction mode. Figure 3.12 shows the 3D transfer function editor. The bottom part controls the scalar opacity mapping and the top part controls the scalar RGB color mapping. The 3D palette contains three regions (from left to right): the hue/saturation display, the value display and the color preview display.



(a) (b)

Figure 3.11: Re-slicing along the current viewing direction.

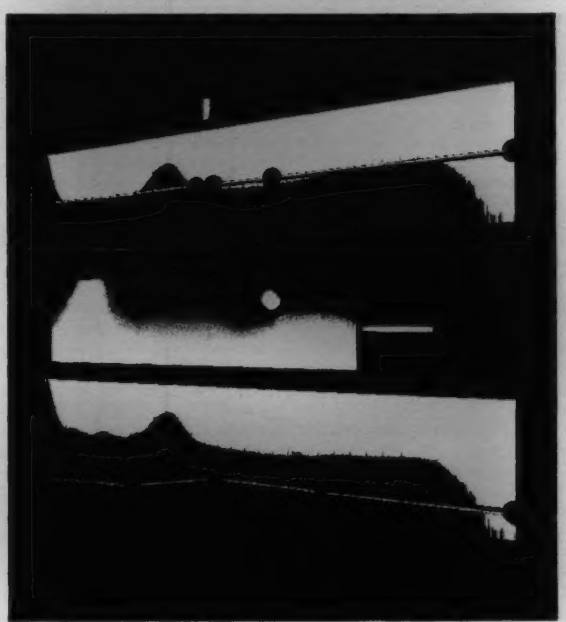


Figure 3.12: The CAVE control panel.

Although the appearance of the editor changes, the operations on the transfer functions remain similar to those in the desktop interaction mode. However, as the widget is placed in a 3D environment, the interaction is a little more complicated than the 2D case. A wand (or joystick) is used instead of a 2D mouse. A red line is drawn from the wand's position and along its direction. This red line serves as a 3D pointer, like the idea employed by Zhang *et al.* [123]. This 3D pointer is used

to manipulate the editor or the displayed volume. A thin bounding box of the transfer function editor is updated whenever the position of the editor changes. The point on the editor that the 3D pointer is aiming at is determined by the intersection between the bounding box and the infinite red line. The interaction style for transforming the displayed volume resembles the joystick-camera style in the desktop interaction mode.

3.5 Software Architecture

MedVis is built using several open-source toolkits and follows the object-oriented programming paradigm. Figure 3.13 gives a high-level block diagram of the visualization pipeline. Medical data acquired from CT or MRI is usually stored in DICOM format. The DICOM image series are read in memory by the *reader* and organized in an internal structure, which is then passed to the *renderer*. The *interactor* handles user input and exchanges states with the renderer to adjust the rendered images. The rest of this section describes the underlying software architecture of MedVis in detail.

Figure 3.13: MedVis's visualization pipeline.

3.5.1 Toolkits Used

Different toolkits are integrated together to support basic visualization and interaction, upon which more sophisticated techniques and algorithms are applied. All the toolkits are cross-platform and widely used in their respective areas.

The Insight Toolkit

The *Insight Toolkit* (ITK) is an application programming interface (API) for image processing, especially the segmentation and registration of medical images [120]. Multiple registration methods and segmentation algorithms are supported. The

data processing pipeline of ITK supports not only an automatic updating mechanism that causes a filter to execute if and only if its input or its internal state changes, but also streaming, the ability to automatically break data into smaller pieces, process the pieces one by one, and reassemble the processed data into a final result [70]. Images with DICOM format are read via the GDCM library (part of ITK) and then treated as ITK internal images. Besides the support for reading DICOM-format images, the ITK pipeline can be seamlessly connected to the VTK (the Visualization Toolkit) pipeline. As such, ITK is chosen as the interface between the DICOM format and MedVis's visualization classes.

The Visualization Toolkit

The *Visualization Toolkit* (VTK) is an open source C++ class library for 3D graphics and visualization [91]. VTK is well-known for its powerful rendering capability: Its rendering model supports 2D, polygonal, volumetric, and texture-based approaches that can be used in any combination [90]. Some of its design concepts are the same as ITK, such as the separation of data objects and process objects, automatic updating mechanism. In addition, object factories and virtual functions are largely utilized to maximize the portability and extensibility, which makes adding new rendering classes (or classes of other purposes) very convenient. Although VTK does not depend on any graphical user interface (GUI), it can be easily integrated with many existing GUI toolkits such as Qt, Tk and MFC. Due to VTK's well-structured visualization pipeline and extensibility, it is chosen as a basic visualization layer, upon which the MedVis's rendering model is built.

The Gimp Toolkit

The *Gimp Toolkit* (GTK+) is a multi-platform toolkit for creating graphical user interfaces, which offers a complete set of 2D widgets [31]. The desktop interface of MedVis is built on it as VTK does not provide any GUI. With *vtkgtk*, an interface for using VTK within a GTK+ widget in the X Window system [34], VTK's rendering output can be redirected into a GtkDrawingArea widget, therefore, a VTK render window can be embedded into a GTK+ render window. GTK+ passes the received user input signal (*e.g.*, clicking a mouse button or pressing a key) to VTK's interactor, where the actual processing takes place. We have extended vtkgtk to work in the Windows environment and added new functions.

VR Juggler

VR Juggler is a virtual platform for the creation and execution of immersive applications, which provides a VR system-independent operating environment [6]. It provides the abstraction of the hardware via the kernel interface and the abstraction of the graphics API via the draw manager. With a proper configuration file, VR Juggler can automatically handle the correct projections for each wall in the CAVE and the synchronization between each cluster node. The CAVE interface of MedVis is built on it. As MedVis's visualization model is based on VTK's pipeline, *vjVTK* [9] is employed to enable the use of VTK within VR Juggler.

The Virtual Reality Peripheral Network

The *Virtual Reality Peripheral Network* (VRPN) provides a device-independent and network-transparent interface to VR peripherals [100]. It supports many devices, including Microsoft joysticks and InterSense motion trackers. VR Juggler has a VRPN driver that can act as an client and access the states of VR peripherals through a VRPN server that runs separately from VR Juggler applications. Therefore, VRPN is used together with VR Juggler to provide a uniform hardware interface for MedVis's CAVE edition.

CMake

CMake is an open source build manager for software projects that allows developers to specify build parameters in a simple configuration file [64]. This file is then used by CMake to generate native makefiles or workspaces for compilers or integrated development environments (IDE) under various operating systems, such as Microsoft Visual Studio under Windows or the GNU Compiler Collection (GCC) under Linux. We use CMake to generate platform and compiler-dependent project files for compiling MedVis in different environments, which enhances the portability of MedVis.

3.5.2 MedVis Kernel Module

The kernel module deals with all VR setup-independent operations. The operations, together with the data structures that they are applied to, are encapsulated in classes.

The DICOM volumes are represented by the vtkDICOMVolume class, which also takes care of rendering the volume into a 3D scene. Figure 3.14 illustrates the structure of the vtkDICOMVolume class. The classes represented by the green boxes are VTK's classes and those represented by the yellow boxes are MedVis's kernel classes[2]. vtkDICOMVolume inherits VTK's vtkVolume. Besides the mappers that vtkVolume supports, vtkDICOMVolume has two additional rendering options: vtkVolmeTextureMapper3DCg, which implements the GPU-based object-order volume rendering algorithm described in Section 3.2.1, and vtkVolumeRayCastMapper with vtkVolumeRayCastJitteredCompositeFunction, which implements the ray casting algorithm with jittered sample intervals described in Section 3.2.1.

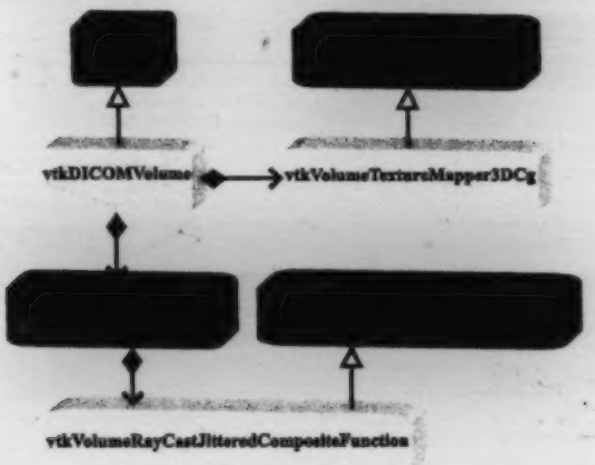


Figure 3.14: The vtkDICOMVolume class.

The manipulation of the transfer functions is carried out via a set of histogram classes, as shown in Figure 3.15. The vtkHistogram class provides basic background histogram rendering, and foreground transfer function rendering and interaction. Its three subclasses, vtkColorHistogram, vtkOpacityHistogram and vtkGradientOpacityHistogram, control the specific rendering and interaction requirements for their respective transfer functions through their own interactor style classes.

[2]All the subsequent figures that illustrate MedVis's classes follow the same mode of representation.

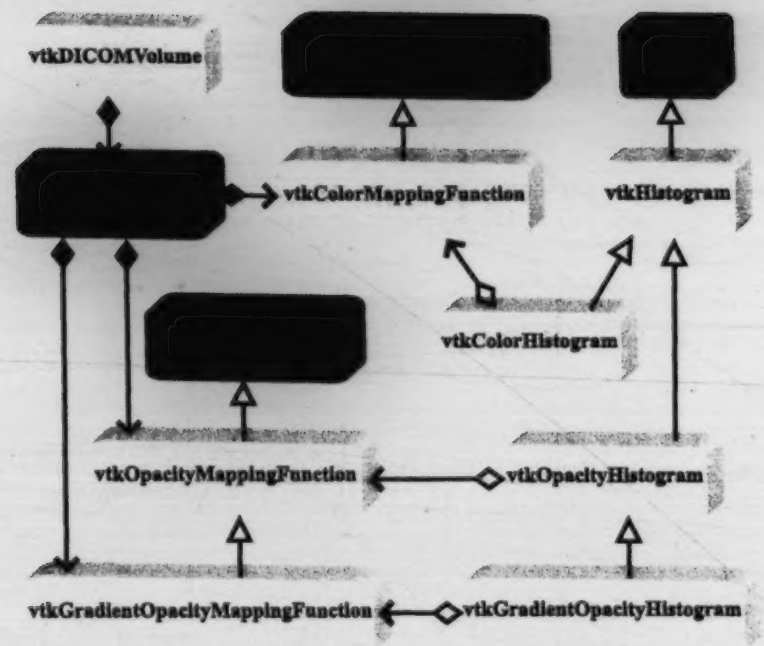


Figure 3.15: The histogram classes.

The vtkDICOMVolume and the histogram classes are connected by several transfer function classes. For instance, the color mapping of vtkDICOMVolume is controlled by vtkColorHistogram via vtkColorMappingFunction. Whenever the vtkColorMappingFunction is changed, the active mapper of vtkDICOMVolume updates the rendering output accordingly.

3.5.3 MedVis Desktop Interface Module

The desktop interface module exports the kernel functions to the PC-based VR systems, like autostereoscopic displays. The time-parallel stereo rendering is implemented by dividing the display window into a left viewport and a right viewport and placing a virtual camera into each viewport. Generating the correct images for each viewport is done in the vtkStereoRenderWindowInteractor class, as shown in Figure 3.16. The classes represented by the blue boxes are MeVi's desktop interface classes. This class also handles the synchronization of the interaction in the two viewports. The four interaction styles mentioned in Section 3.4.1 can be switched at runtime via the vtkStereoInteractorStyleSwitch class.

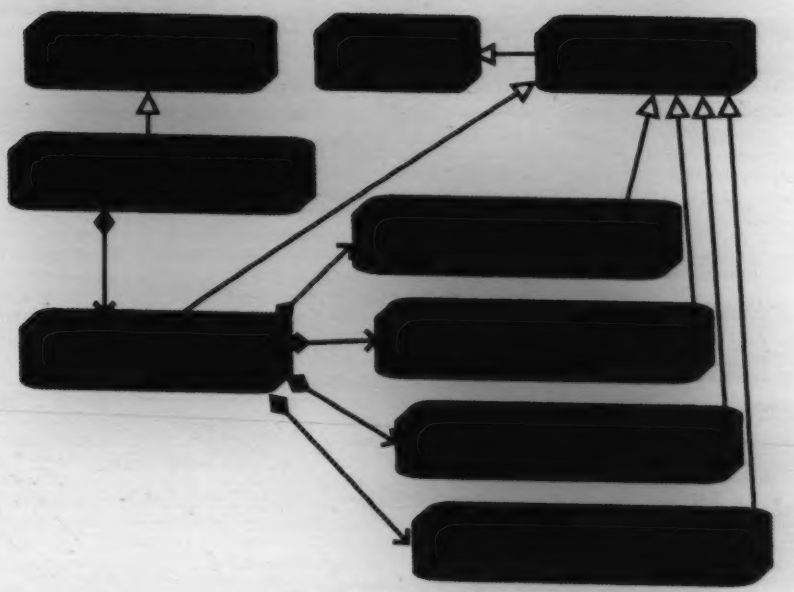


Figure 3.16: The vtkStereoRenderWindowInteractor class.

The desktop GUI is built upon GTK+. The vtkGtkStereoRenderWindowInteractor class embeds a vtkRenderWindow with two viewports (left and right) into a GTK+ window. As shown in Figure 3.17, it is a subclass of the vtkStereoRenderWindowInteractor, which synchronizes the interaction in the two viewports, and the vtkGtkRenderWindowInteractor, which actually connects the kernel rendering output with GTK+'s drawing area and passes the user input signals intercepted by GTK+ to the kernel.

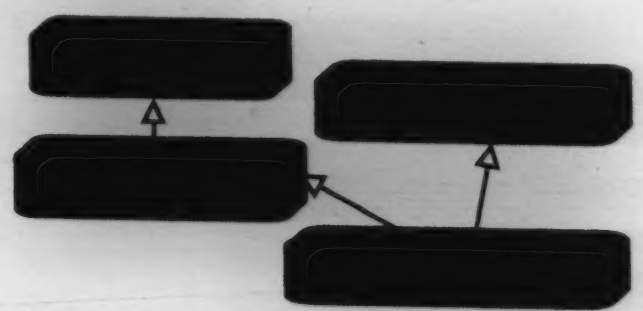


Figure 3.17: The vtkGtkStereoRenderWindowInteractor class.

The abstraction of a high-level GUI class, together with the multiple inheri-

tance, facilitates the addition of basic interaction functions without modification to the kernel module and subclasses that provide specific interactions. With the same concept, the kernel histogram classes are extended to form a GTK+-style transfer function editor. For instance, the relationship between the vtkGtkColorHistogram class and its superclasses is illustrated in Figure 3.18. The slice viewer is implemented in the vtkGtkSliceViewer2 class (Figure 3.19) with the vtkGtkImageViewer2 class displaying the current selected slice.

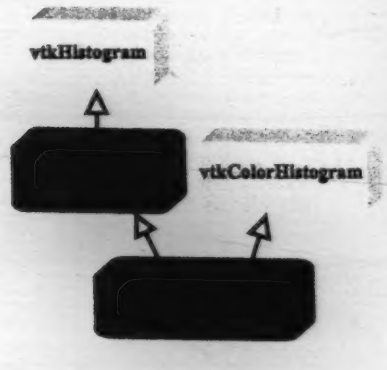


Figure 3.18: The vtkGtkColorHistogram class.

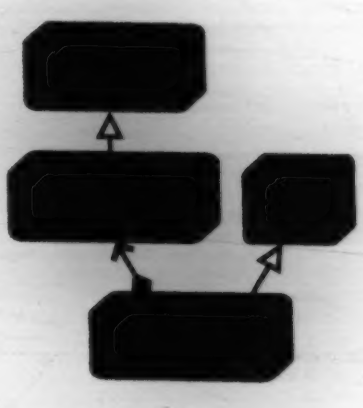


Figure 3.19: The vtkGtkSliceViewer2 class.

3.5.4 MedVis CAVE Interface Module

The CAVE interface module extends the kernel visualization pipeline into the immersive VE, CAVE. MedVis's CAVE GUI is built upon VR Juggler and vjVTK. VR Juggler uses the application object instead of the traditional *main()* function, and the VR Juggler kernel schedules the application by calling the object's interface meth-

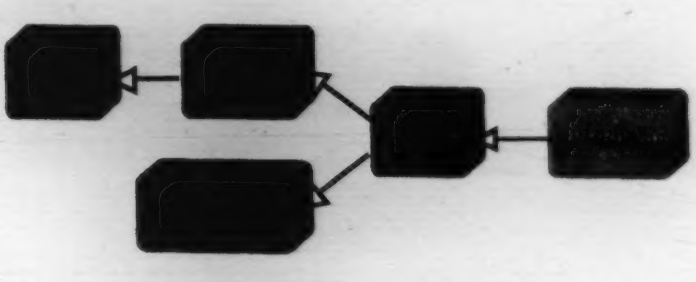


Figure 3.20: The vtkVRJApp class.

ods. MedVis's CAVE application interface is defined in the vtkVRJApp class. The relationship between vtkVRJApp and its superclasses is illustrated in Figure 3.20. The classes represented by the gray, brown and orange boxes are MedVis's CAVE interface classes, vjVTK classes and VR Juggler classes respectively.

The base application object interface is defined in the vrj::App class, which is in charge of initializing an application and invoking the frame functions during the kernel frame loop. vrj::App's subclass vrj::GlApp defines the draw manager interface that allows the rendering of OpenGL graphics. The VTKApp_mixin class creates a VTK rendering environment with derived vtkRenderWindow, vtkRenderer and vtkCamera classes. The VTKApp class, derived from both vrj::GlApp and VTKApp_mixin, renders a VTK scene into the VTK context setup by VTKApp_mixin in the VR Juggler draw manager. The vtkVRJApp class utilizes the interfaces and functions provided by VTKApp, and connects the MedVis's rendering classes and interaction classes together to work correctly in the CAVE environment.

The vtkVRJDICOMVolume class (Figure 3.21), derived from vtkDICOMVolume, uses vtkVRJVolumeTextureMapper3DCg to provide the hardware-accelerated volume rendering. The vtkVRJVolumeTextureMapper3DCg class inherits all the functions and properties of vtkVolumeTextureMapper3DCg except that the head position is now determined by a position proxy that transfers the position data captured in real world to VR Juggler-based applications.

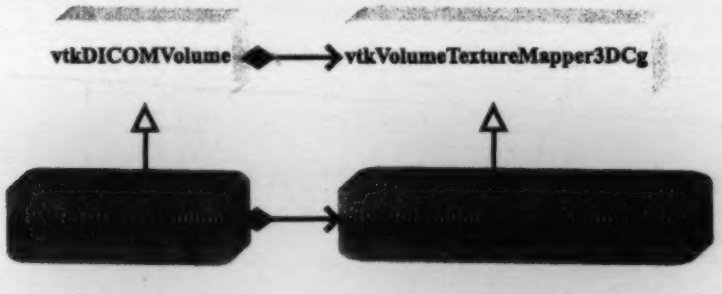


Figure 3.21: The vtkVRJDICOMVolume class.

The transfer function editor in the CAVE is a 3D widget that provides the user interface for real-time modification of the transfer functions. The structure of the vtkVRJTransferFunctionEditor class is shown in Figure 3.22. The vtkVRJOpacityHistogram class controls the editing of the opacity mapping, while the vtkVR-

JColorHistogram class, together with the vtkVRJColorSelection class that holds the color palette, controls the editing of the RGB color mapping. The vtkVRJOpacityHistogram and vtkVRJColorHistogram classes both have a common superclass, vtkVRJHistogram, which serves a basic interface between MedVis kernel histogram classes and VR Juggler. Their the other respective superclasses (vtkOpacityHistogram and vtkColorHistogram) are the classes that actually provide the specific interaction cababilities.

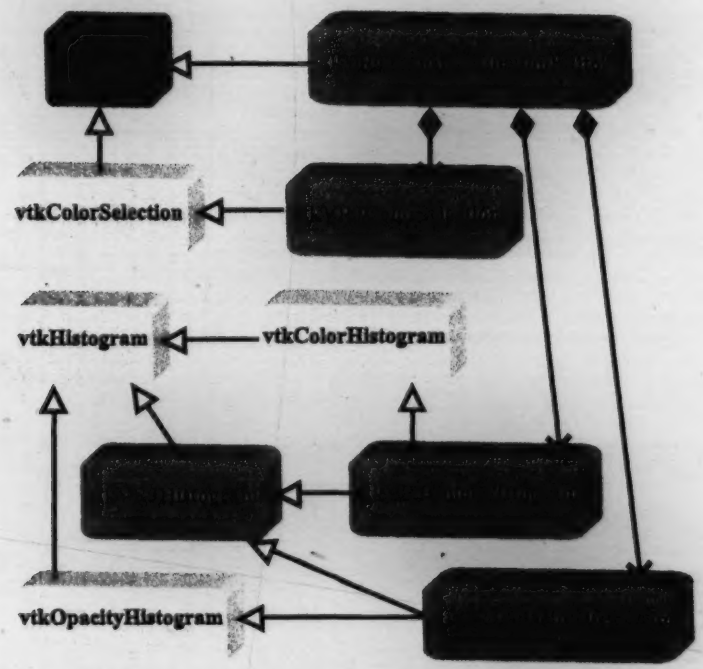


Figure 3.22: The vtkVRJTransferFunctionEditor class.

Chapter 4

Performance and Results

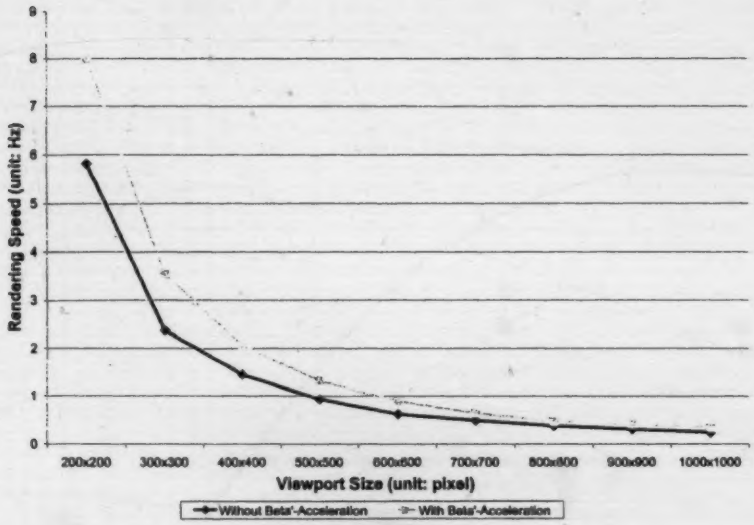
4.1 System Performance for Rendering

The system is tested on a dual-core 2.0GHz computer running Windows XP with a 256MB-memory NVIDIA GeForce 7800 GTX graphics card. The data used for testing is a medium-size (512x512x181) CT-scan of the pelvic region.

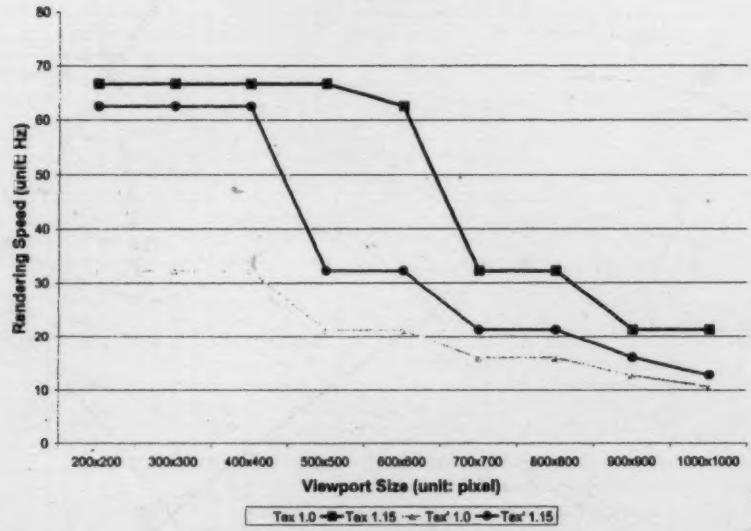
Table 4.1: The rendering times using different acceleration techniques.

Viewport Size (unit: pixel)	Rendering Time (unit: second)					
	Without β'	With β'	Tex' 1.0	Tex' 1.15	Tex 1.0	Tex 1.15
200x200	0.172	0.125	0.031	0.016	0.015	0.015
300x300	0.422	0.281	0.031	0.016	0.015	0.015
400x400	0.688	0.484	0.031	0.016	0.015	0.015
500x500	1.078	0.750	0.047	0.031	0.031	0.015
600x600	1.641	1.125	0.047	0.031	0.031	0.016
700x700	2.078	1.562	0.062	0.047	0.047	0.031
800x800	2.704	2.187	0.062	0.047	0.047	0.031
900x900	3.391	2.469	0.078	0.062	0.062	0.047
1000x1000	4.312	3.062	0.094	0.078	0.078	0.047

Software-based ray casting provides high quality images, but only with small viewports or for small data sets it can maintain an acceptable rendering speed, even with the proposed β'-acceleration. With the transfer functions shown at Figure 4.2(g), the rendering times using software ray casting with both early ray termination and β'-acceleration and with only early ray termination are enumerated in the first two columns of Table 4.1. Figure 4.1(a) depicts the two cases' performance curves with respect to the viewport size. The x-axis is the size of the viewport in pixels and the y-axis is the rendering speed in hertz. The dark gray line denotes the performance of the method without β'-acceleration, and the other line denotes



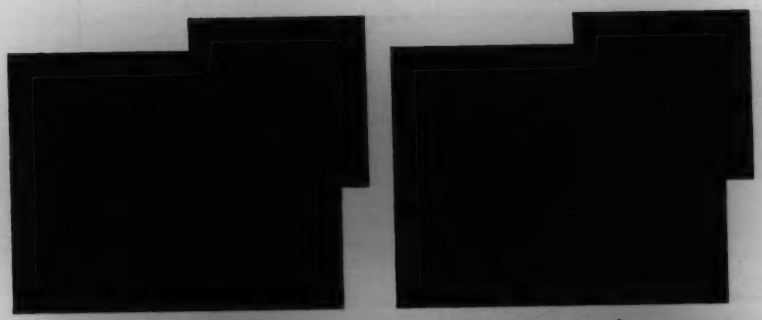
(a) Ray Casting.



(b) Object-Order.

Figure 4.1: The comparison of the rendering speeds using different acceleration techniques.

the performance of the one with β'-acceleration. On the average, software ray casting with both early ray termination (Γ=0.02) and β'-acceleration (Γ'=0.6 and f=0.1) takes 28% less time than that with only early ray termination (Γ=0.02). The resulting images are shown in Figure 4.2(a)(b). There is no noticeable difference between

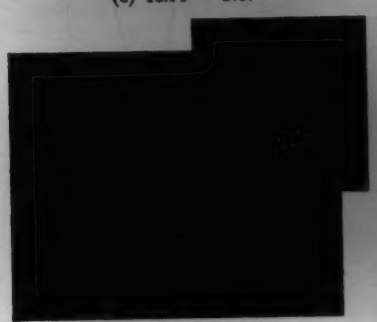


(a) Without β'.

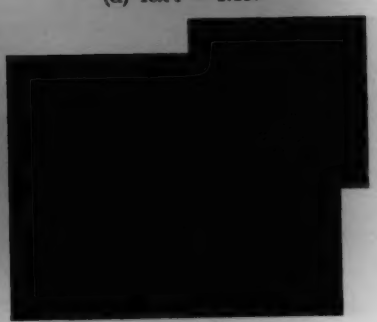
(b) With β'.

(c) Tex $F = 1.0$.

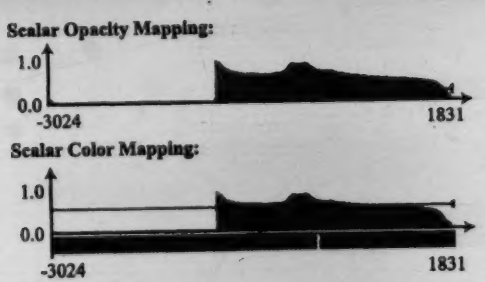
(d) Tex $F = 1.15$.



(e) Tex′ $F = 1.0$.



(f) Tex′ $F = 1.15$.



(g) Transfer functions.

Figure 4.2: Volume rendering results of a CT-scanned pelvic region in MedVis.

these two images.

High-quality images and interactive rendering speed are both achieved by exploiting the processing power of the GPU. The rendering times under four different conditions are enumerated in Table 4.1. *Tex' 1.0* denotes no acceleration; *Tex' 1.15* denotes adaptive sample interval with interval scale factor F=1.15; *Tex 1.0* denotes only with vertex shader acceleration; *Tex 1.15* denotes with both acceleration techniques and F=1.15. Figure 4.1(b) gives a comparison of the performance curves under the four different conditions. In all cases, the rendering speed decreases as the viewport grows, but even for the 1000x1000 viewport the rendering times are below 0.1 second, *i.e.*, the rendering speeds are above the psycho-physical limit of 10 Hz. With only adaptive sample interval enabled, when F=1.15, we get an average 33% speedup. With only vertex shader acceleration enabled, the algorithm's performance is almost the same as Tex' 1.15. With both acceleration techniques enabled, when F=1.15, an average 53% speedup is achieved with respect to the Tex' 1.0 case and an average 28% speedup is achieved with respect to the Tex' 1.15 case. Since stereo rendering is necessary, the actual rendering time is doubled. However, an interactive frame rate is still maintained. Stereo volume rendering of this data

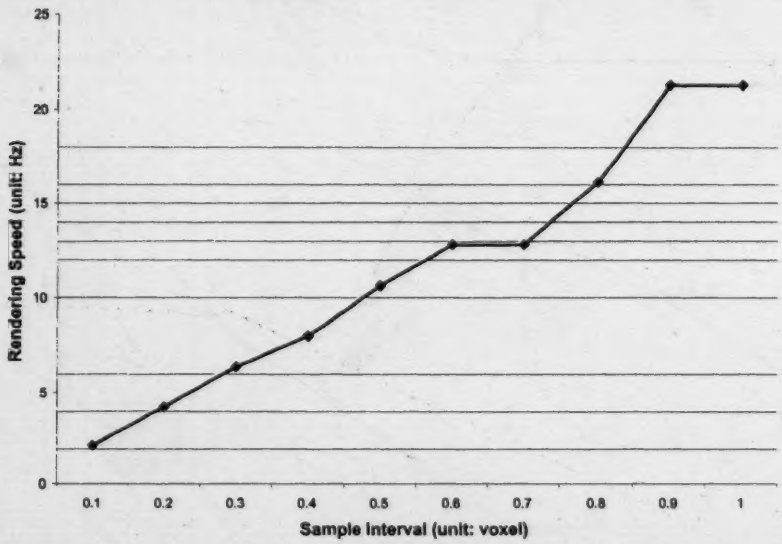
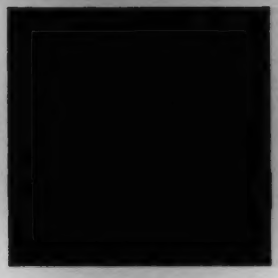


Figure 4.3: The rendering speed of MedVis's GPU-based algorithm with different sample intervals.

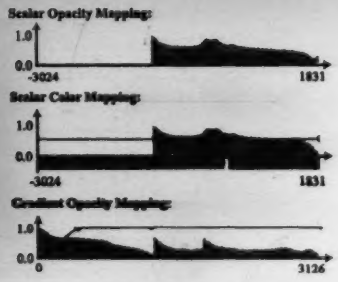
(a) $s = 0.1$. (b) $s = 0.2$. (c) $s = 0.3$.

(d) $s = 0.4$. (e) $s = 0.5$. (f) $s = 0.6$.

(g) $s = 0.7$. (h) $s = 0.8$. (i) $s = 0.9$.



(j) $s = 1.0$.



(k) Transfer functions.

Figure 4.4: Volume rendering results of a CT-scanned pelvic region in MedVis with different sample intervals.

set into two 512x512 viewports using our method with F=1.0 has an average frame rate of 17Hz. If F=1.15, the average frame rate is about 33Hz. The final images are shown in Figure 4.2(c)-(f), together with the images produced by software ray casting. From these images, no significant difference can be observed between the image quality of image-order methods and that of object-order methods, as long as the original data set is at high resolution.

Normally, the rendering speed increases as the sample interval increases. Figure 4.3 depicts the change of the rendering speed with respect to the sample interval (from 0.1 to 1.0). The data is collected based on MedVis's GPU-based algorithm with only inside-vertex-shader polygon generation acceleration enabled. The x-axis is the sample interval in voxels and the y-axis is the rendering speed in hertz. The applied transfer functions are shown in Figure 4.4(k). When s=0.1, the rendering speed is lowest, *i.e.*, 0.453 second to render a single image; but when s decreases to 0.9 or 1.0, the rendering speed increases to 21.28 Hz, which is about 10 times faster than s=0.1. However, the image quality decreases as well, as shown in Figure 4.4(a)-(j). There are noticeable differences when s increases from 0.1 to 0.3. However, there is no noticeable difference when s increases from 0.3 to 1.0. This indicates that when sample interval has reached a threshold (not necessarily a large threshold), an even larger sample interval can be used to enhance the rendering speed with almost no degradation of the image quality. The gradient opacity mapping is applied in addition to the scalar color-opacity mapping. This enables the extraction of material boundaries. At Figure 4.4, the skin (a blue thin 3D surface) is separated from other tissues.

4.2 Results for the Desktop Version

Figure 4.5 shows the rendering result of a CT-scanned abdomen (data size: 512x512x333) in MedVis desktop version. The left and right images shown in Figure 3.6 are merged together to form a 3D image. The user can observe the rendered volume from any position, either inside the volume or outside the volume by transforming the volume with a 2D mouse. The transfer functions and other control parameters can be adjusted by the user using a 2D mouse at runtime, and the changes to the volume are applied in realtime.

Figure 4.5: Stereo volume rendering of a CT-scanned abdomen in the MedVis desktop version.

4.3 Results for the CAVE Version

Figure 4.6(a)(b) show the rendering results of an MRI-scanned heart in MedVis CAVE version. Multiple users can stay in the CAVE simultaneously to analyze the data. The displayed volume can be transformed using the wand, or the user can walk around to observe the data from different directions. The transfer function editor (shown in Figure 4.6(a)) can be turned on/off (shown/hidden) at any time, and it can be dragged by the wand to be placed in a convenient position. Like in the desktop version, the changes of the transfer functions are applied to the volume in realtime, *i.e.*, the user can get the visual feedback of the influence of the current transfer functions immediately. Figure 4.6(b) shows the visually segmented heart.

(a)

(b)

Figure 4.6: Volume rendering of an MRI-scanned heart in the MedVis CAVE version.

Chapter 5

Conclusion and Future Work

This thesis presents the Medical Visualizer, a VR system for visualizing volumetric medical data in various VR systems, ranging from non-immersive to immersive systems. Our main goal is to provide radiologists more help in understanding the data produced by CT and MRI. Real-time high-quality stereo volume rendering and interactive manipulation of the color and transparency classifications are supported for effectively analyzing the fine details in the data sets. Several volume rendering acceleration techniques are proposed for medical data visualization. β'-acceleration enhances the rendering speed of software-based ray casting using voxels' opacity information, while vertex shader proxy polygon generation and adaptive sample interval improve the performance of traditional hardware-accelerated object-order volume rendering. Remarkable speedups are observed from experiments on average-size medical data sets. In addition, since only an autostereoscopic display and a standard PC are required, MedVis desktop version can be easily incorporated into radiologists's daily workflow for pre-surgical planning or diagnosis. With MedVis CAVE version, radiologists can explore the volumetric data sets in a more natural way and easily get a better understanding of the 3D structure. The VR setup-dependent functions are separated from the kernel module, which only deals with volume rendering and interactive classification. The user interface modules handle stereo rendering and the connection between the GUI and the kernel module. Due to the modular design, MedVis is also easily extensible to other virtual environment modalities, and new functions can be incorporated rapidly.

5.1 Future Work

What is now required is to perform a user study to demonstrate the advantages of our system relative to the traditional film-based approach. This is key for acceptability of this system in the radiology community as we are competing with a well established practice and it is up to us to prove that this approach does improve the effectiveness of radiologists to analyze their data sets. Furthermore, we are also exploring more efficient and effective rendering algorithms using GPU clusters to handle larger and larger data sets produced by doppler MRI and temporal CT. In addition, based on modular design concept, we will extend MedVis to work in other VR setups (*e.g.*, with haptic feedback) that may provide more help to radiologists in understanding the data sets.

Bibliography

[1] Advanced Visual Systems Inc. AVS/Express. http://www.avs.com/.

[2] M. Agrawala, A. C. Beers, I. McDowall, B. Fröhlich, M. Bolas, and P. Hanrahan. The two-user Responsive Workbench: support for collaboration through individual views of a shared space. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 327–332, 1997.

[3] K. Akeley. Realityengine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, 1993.

[4] J. Arvo and K. Novins. Iso-contour volume rendering. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 115–122, 1994.

[5] T. Balogh. The HoloVizio system. In *Proceedings of SPIE, Stereoscopic displays and virtual reality systems XIII*, volume 6055, pages 279–290, 2006.

[6] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *VR '01: Proceedings of the Virtual Reality 2001 Conference*, pages 89–97, 2001.

[7] A. P. D. Binotto, J. L. D. Comba, and C. M. D. Freitas. Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *PVG '03: Proceedings of the 2003 IEEE symposium on Parallel and large-data visualization and graphics*, pages 69–75, 2003.

[8] G. Bishop and H. Fuchs. Research directions in virtual environments: report of an NSF invitational workshop, march 23-24, 1992, university of north carolina at chapel hill. *ACM SIGGRAPH Computer Graphics*, 26(3):153–177, 1992.

[9] K. Blom. vjVTK: a toolkit for interactive visualization in virtual reality. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 17–19, 2006.

[10] F. P. Brooks. What's real about virtual reality? *IEEE Computer Graphics and Applications*, 19(6):16–27, 1999.

[11] S. Bryson. Virtual reality in scientific visualization. *Communications of the ACM*, 39(5):62–71, 1996.

[12] S. Bryson and M. Gerald-Yamasaki. The distributed virtual windtunnel. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 275–284, 1992.

[13] G. C. Burdea and P. Coiffet. *Virtual reality technology*. Wiley-IEEE Press, second edition, 2003.

[14] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, 1994.

[15] O. Cakmakci and J. Rolland. Head-worn displays: a review. *Journal of Display Technology*, 2(3):199–216, 2006.

[16] E. E. Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah, 1974.

[17] F. C. Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, 1984.

[18] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142, 1993.

[19] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The CAVE: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72, 1992.

[20] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. L. Dawe, and M. D. Brown. The ImmersaDesk and Infinity Wall projection-based virtual reality displays. *ACM SIGGRAPH Computer Graphics*, 31(2):46–49, 1997.

[21] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 69–77, 1998.

[22] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, 1992.

[23] Dimension Technologies Inc. Virtual Window. http://www.dti3d.com/.

[24] N. A. Dodgson, J. R. Moore, S. R. Lang, G. J. Martin, and P. M. Canepa. A 50" time-multiplexed autostereoscopic display. In *Proceedings of SPIE, Stereoscopic displays and virtual reality systems VII*, volume 3957, pages 177–183, 2000.

[25] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, 1988.

[26] T. T. Elvins. A survey of algorithms for volume visualization. *ACM SIGGRAPH Computer Graphics*, 26(3):194–201, 1992.

[27] A. Entezari, R. Scoggins, T. Möller, and R. Machiraju. Shading for fourier volume rendering. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 131–138, 2002.

[28] G. E. Favalora, J. Napoli, D. M. Hall, R. K. Dorval, M. Giovinco, M. J. Richmond, and W. S. Chun. 100-million-voxel volumetric display. In *Proceedings of SPIE, Cockpit displays IX: Displays for defense applications*, volume 4712, pages 300–312, 2002.

[29] P. Figueroa, M. Green, and H. J. Hoover. InTml: a description language for VR applications. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, pages 53–58, 2002.

[30] A. S. Forsberg, D. H. Laidlaw, A. van Dam, R. M. Kirby, G. E. Karniadakis, and J. L. Elion. Immersive virtual reality for visualizing flow through an artery. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 457–460, 2000.

[31] GNOME Foundation. The Gimp Toolkit. http://www.gtk.org/.

[32] B. Fröhlich, S. Barrass, B. Zehner, J. Plate, and M. Göbel. Exploring geo-scientific data in virtual environments. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 169–173, 1999.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Inc., first edition, 1995.

[34] D. Grobgeld. vtkgtk. http://imagic.weizmann.ac.il/dov/freesw/gtk/vtkgtk/.

[35] S.-Y. Guan and R. G. Lipes. Innovative volume rendering using 3d texture mapping. In *Proceedings of SPIE, Medical imaging 1994: image capture, formatting, and display*, volume 2164, pages 382–392, 1994.

[36] M. Hadwiger, C. Berger, and H. Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardwarem. In *VIS '03: Proceedings of the conference on Visualization '03*, pages 301–308, 2003.

[37] M. Halle. Autostereoscopic displays and computer graphics. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 104, 2005.

[38] N. Hata, T. Wada, T. Chiba, Y. Tsutsumi, Y. Okada, and T. Dohi. Three-dimensional volume rendering of fetal MR images for the diagnosis of congenital cystic adenomatoid malformation. *Academic Radiology*, 10(3):309–312, 2003.

[39] H. Hauser, L. Mroz, G. I. Bischi, and M. E. Groller. Two-level volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):242–252, 2001.

[40] P. S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.

[41] L. F. Hodges. Tutorial: time-multiplexed stereoscopic computer graphics. *IEEE Computer Graphics and Applications*, 12(2):20–30, 1992.

[42] V. Jaswal. CAVEvis: distributed real-time visualization of time-varying scalar and vector fields using the CAVE virtual reality theater. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 301–308, 1997.

[43] A. Johnson, J. Leigh, P. Morin, and P. Van Keken. GeoWall: stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 2(6):10–14, 2006.

[44] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH Computer Graphics*, 18(3):165–174, 1984.

[45] A. C. Kak and M. Slaney. *Principles of computerized tomographic imaging*. Society for Industrial and Applied Mathematics, 2001.

[46] A. E. Kaufman. Volume visualization. *ACM Computing Surveys*, 28(1):165–167, 1996.

[47] H. Kaufmann, D. Schmalstieg, and M. Wagner. Construct3D: a virtual reality application for mathematics and geometry education. *Education and Information Technologies*, 5(4):263–276, 2000.

[48] Kitware Inc. VolView. http://www.kitware.com/.

[49] Kitware Inc., Sandia National Laboratories, Los Alamos National Laboratory, Army Research Laboratory, and CSimSoft. ParaView. http://www.paraview.org/.

[50] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 255–262, 2001.

[51] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[52] J. Kniss, J. P. Schulze, U. Wössner, P. Winkler, U. Lang, and C. Hansen. Medical applications of multi-field volume rendering and VR techniques. In *Proceedings of the Joint Eurographics-IEEE TCVG symposium on Visualization*, pages 249–254, 2004.

[53] A. Kratz, M. Hadwiger, R. Splechtna, A. Fuhrmann, and K. Bühler. GPU-based high-quality volume rendering for virtual environments. In *Proceedings of international workshop on Augmented environments for medical imaging and computer aided surgery*, 2006.

[54] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003*, pages 287–292, 2003.

[55] W. Krüger, C.-A. Bohn, B. Fröhlich, H. Schüth, W. Strauss, and G. Wesche. The Responsive Workbench: a virtual work environment. *Computer*, 28(7):42–48, 1995.

[56] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, 1994.

[57] R. J. Lapeer, R. S. Rowland, and M. S. Chen. Pc-based volume rendering for medical visualisation and augmented reality based surgical navigation. In *IV '04: Proceedings of the 8th international conference on Information visualisation*, pages 67–72, 2004.

[58] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 285–288, 1991.

[59] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[60] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[61] M. Levoy, H. Fuchs, S. M. Pizer, J. Rosenman, E. L. Chaney, G. W. Sherouse, V. Interrante, and J. Kiel. Volume rendering in radiation treatment planning. In *Proceedings of the first conference on Visualization in biomedical computing*, pages 22–25, 1990.

[62] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, 1987.

[63] T. Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, 1993.

[64] K. Martin and B. Hoffman. *Mastering Cmake: a cross-platform build system.* Kitware Inc., 2006.

[65] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in scientific computing. *ACM SIGGRAPH Computer Graphics*, 21(6), 1987.

[66] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90, 2000.

[67] B. Mora, J. P. Jessel, and R. Caubet. A new object-order ray-casting algorithm. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 203–210, 2002.

[68] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[69] H. Nagahara, Y. Yagi, and M. Yachida. Wide field of view catadioptrical head-mounted display. In *Proceedings of the 2003 IEEE/RSJ international conference on Intelligent robots and systems*, volume 4, pages 3738–3743, 2003.

[70] L. Ibá nez and W. Schroeder. *The ITK software guide.* Kitware Inc., second edition, 2005.

[71] D. Pape, J. Anstey, and G. Dawe. Low-cost projection-based virtual reality display. In *Proceedings of SPIE, Stereoscopic displays and virtual reality systems IX*, volume 4660, pages 483–491, 2002.

[72] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 233–238, 1998.

[73] R. Pausch. Virtual reality on five dollars a day. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 265–270, 1991.

[74] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 251–260, 1999.

[75] H. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 47–54, 1996.

[76] W. Qi, R. M. Taylor II, C. G. Healey, and J.-B. Martens. A comparison of immersive HMD, fish tank VR and fish tank with haptics displays for volume visualization. In *APGV '06: Proceedings of the 3rd symposium on Applied perception in graphics and visualization*, pages 51–58, 2006.

[77] P. J. Rajlich. An object oriented approach to developing visualization tools portable across desktop and virtual environments. Master's thesis, University of Illinois at Urbana-Champaign, 1998.

[78] D. Rantzau, K. Frank, U. Lang, D. Rainer, and U. Wössner. COVISE in the CUBE: an environment for analyzing large and complex simulation data. In *Proceedings of the 2nd Workshop on Immersive Projection Technology*, 1998.

[79] A. G. Requicha. Representations for rigid solids: theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.

[80] C. Rezk-Salama and A. Kolb. A vertex program for efficient box-plane intersection. In *Proceedings of the 10th international fall workshop on Vision, modeling and visualization*, 2005.

[81] W. Ribarsky, J. Bolter, A. Op den Bosch, and R. van Teylingen. Visualization and analysis using virtual reality. *IEEE Computer Graphics and Applications*, 14(1):10–12, 1994.

[82] G. Riva. Applications of virtual environments in medicine. *Methods of Information in Medicine*, 42(5):524–534, 2003.

[83] J. Rohlf and J. Helman. IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, 1994.

[84] D. Salzman and J. von Neumann. Visualization in scientific computing: summary of an nsf-sponsored panel report on graphics, image processing, and workstations. *International Journal of High Performance Computing Applications*, 1:106–108, 1987.

[85] D. J. Sandin, T. Margolis, J. Ge, J. Girado, T. Peterka, and T. A. DeFanti. The Varrier™ autostereoscopic virtual reality display. *ACM Transactions on Graphics*, 24(3):894–903, 2005.

[86] H. Scharsach. Advanced GPU raycasting. In *Proceedings of CESCG 2005*, pages 69–76, 2005.

[87] H. Scharsach, M. Hadwiger, A. Neubauer, S. Wolfsberger, and K. Bühler. Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *Proceedings of Eurovis/IEEE-VGTC symposium on Visualization*, 2006.

[88] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, L. M. Encarnação, M. Gervautz, and W. Purgathofer. The studierstube augmented reality project. *Presence: Teleoperators and Virtual Environments*, 11(1):33–54, 2002.

[89] J. Schneider and R. Westermann. Compression domain volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003*, pages 293–300, 2003.

[90] W. J. Schroeder, L. S. Avila, and W. Hoffman. Visualizing with VTK: a tutorial. *IEEE Computer Graphics and Applications*, 20(5):20–27, 2000.

[91] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 93–102, 1996.

[92] J. P. Schulze and U. Lang. The parallelization of the perspective shear-warp volume rendering algorithm. In *EGPGV '02: Proceedings of the fourth Eurographics workshop on Parallel graphics and visualization*, pages 61–69, 2002.

[93] J. P. Schulze, U. Wössner, S. P. Walz, and U. Lang. Volume rendering in a virtual environment. In *Proceedings of the 5th IPTW and Eurographics Virtual environments*, pages 187–198, 2001.

[94] Sensics Inc. piSight. http://www.sensics.com/.

[95] W. Shroeder, K. Martin, and B. Lorensen. *The visualization toolkit: an object-oriented approach to 3D graphics*. Pearson Education, Inc., fourth edition, 2006.

[96] A. Sourin, O. Sourina, and H. T. Sen. Virtual orthopedic surgery training. *IEEE Computer Graphics and Applications*, 20(3):6–9, 2000.

[97] A. Sullivan. Depthcube solid-state 3d volumetric display. In *Proceedings of SPIE, Stereoscopic displays and virtual reality systems XI*, volume 5291, pages 279–284, 2004.

[98] I. E. Sutherland. The ultimate display. In *Proceedings of the IFIP Congress*, volume 2, pages 506–508, 1965.

[99] I. E. Sutherland. A head-mounted three dimensional display. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 33, pages 757–764, 1968.

[100] R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. Vrpn: a device-independent, network-transparent VR peripheral system. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61, 2001.

[101] M. A. Teitel. The Eyephone: a head-mounted stereo display. In *Proceedings of SPIE, Stereoscopic displays and applications*, volume 1256, pages 168–171, 1990.

[102] T. Totsuka and M. Levoy. Frequency domain volume rendering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 271–278, 1993.

[103] C. Upson and M. Keeler. V-buffer: visible volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 59–64, 1988.

[104] A. van Dam, A. S. Forsberg, D. H. Laidlaw, J. J. LaViola, and R. M. Simpson. Immersive VR for scientific visualization: a progress report. *IEEE Computer Graphics and Applications*, 20(6):26–52, 2000.

[105] T. van der Schaaf, D. M. Germans, M. Koutek, and H. E. Bal. ICWall: a calibrated stereo tiled display from commodity components. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 289–296, 2006.

[106] A. Van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 23–30, 1996.

[107] I. Viola, A. Kanitsar, and M. E. Gröller. GPU-based frequency domain volume rendering. In *SCCG '04: Proceedings of the 20th spring conference on Computer graphics*, pages 55–64, 2004.

[108] C. Ware, K. Arthur, and K. S. Booth. Fish tank virtual reality. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 37–42, 1993.

[109] Z. Wartell, L. F. Hodges, and W. Ribarsky. Characterizing image fusion techniques in stereoscopic htds. In *Proceedings of Graphics interface 2001*, pages 223–232, 2001.

[110] M. A. Westenberg and J. B. T. M. Roerdink. Frequency domain volume rendering by the wavelet x-ray transform. *IEEE Transactions on Image Processing*, 9(7):1249–1261, 2000.

[111] L. Westover. Interactive volume rendering. In *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 9–16, 1989.

[112] L. Westover. Footprint evaluation for volume rendering. *ACM SIGGRAPH Computer Graphics*, 24(4):367–376, 1990.

[113] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. *ACM SIGGRAPH Computer Graphics*, 25(4):275–284, 1991.

[114] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM SIGGRAPH Computer Graphics*, 11(3):201–227, 1992.

[115] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5):19–28, 1992.

[116] R. Yagel and A. Kaufman. Template-based volume viewing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS '92)*, volume 11, pages 153–167, 1992.

[117] R. Yagel, A. Kaufman, and Q. Zhang. Realistic volume imaging. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 226–231, 1991.

[118] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 62–69, 1993.

[119] T. S. Yoo, editor. *Insight into images: principles and practice for segmentation, registration, and image analysis*. A K Peters, Ltd., first edition, 2004.

[120] T. S. Yoo, M. J. Ackerman, W. E. Lorensen, W. Schroeder, V. Chalana, S. Aylward, D. Metaxes, and R. Whitaker. Engineering and algorithm design for an image processing API: a technical report on ITK - the Insight Toolkit. In *Proceedings of Medicine Meets Virtual Reality*, pages 586–592, 2002.

[121] A. Yoshida, J. P. Rolland, and J. H. Reif. Design and applications of a high-resolution insert head-mounted-display. In *VRAIS '95: Proceedings of the virtual reality annual international symposium*, volume 4, pages 84–93, 1995.

[122] R. Zajtchuk and R. M. Satava. Medical applications of virtual reality. *Communications of the ACM*, 40(9):63–64, 1997.

[123] S. Zhang, C. Demiralp, D. F. Keefe, M. DaSilva, D. H. Laidlaw, B. D. Greenberg, P. J. Basser, C. Pierpaoli, E. A. Chiocca, and T. S. Deisboeck. An immersive virtual environment for DT-MRI volume visualization applications: a case study. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 437–440, 2001.

Appendix A

Cg Code

A.1 Vertex Shader Code

```
void V_ComputingPolygons(
    int2 pos : POSITION,
    uniform float dPlaneStart,
    uniform float4x4 modelViewProj,
    uniform float frontIndex,
    uniform float dPlaneIncr,
    uniform float3 vecVertices[11],
    uniform float nSequence[64],
    uniform float v1[24],
    uniform float v2[24],
    out float4 vertex : POSITION,
    out float3 texCoord0 : TEXCOORD0)
{
    float dPlaneDist = dPlaneStart + pos.y * dPlaneIncr;
    float3 position;
    float3 vecV1, vecV2, vecDir;
    float denom, lambda;
    int i = 0, vidx1, vidx2;
    for (i = 0; i < 4; i++)
    {
        vidx1 = int(nSequence[int(frontIndex * 8 + v1[pos.x * 4 + i])]);
        vidx2 = int(nSequence[int(frontIndex * 8 + v2[pos.x * 4 + i])]);
        vecV1 = vecVertices[vidx1];
        vecV2 = vecVertices[vidx2];
        vecDir = vecV2 - vecV1;
        denom = dot(vecDir, vecVertices[8]);
        lambda =
            (denom != 0)?(dPlaneDist - dot(vecV1, vecVertices[8]))/denom:-1.0;
        if (lambda >= 0.0 && lambda <= 1.0)
        {
            position = vecV1 + lambda * vecDir;
            break;
```

```
        }
    }
    vertex = mul(modelViewProj, float4(position, 1.0));
    if (vecDir.x == 0)
        texCoord0.x = (vecV1.x == vecVertices[0].x)?1:0;
    else
        texCoord0.x = (vecDir.x > 0)?lambda:(1 - lambda);
    if (vecDir.y == 0)
        texCoord0.y = (vecV1.y == vecVertices[0].y)?1:0;
    else
        texCoord0.y = (vecDir.y > 0)?lambda:(1 - lambda);
    if (vecDir.z == 0)
        texCoord0.z = (vecV1.z == vecVertices[0].z)?1:0;
    else
        texCoord0.z = (vecDir.z > 0)?lambda:(1 - lambda);
    texCoord0 = texCoord0 * vecVertices[9] + vecVertices[10];
}
```

A.2 Fragment Shader Code

```
void F_Sampling(
    float3 texCoord0 : TEXCOORD0,
    out float4 color : COLOR,
    uniform sampler3D volume : TEX0,
    uniform sampler2D colorLookup : TEX1)
{
    color = tex2D(colorLookup, tex3D(volume, texCoord0).ar);
}
```

Appendix B

GPU-Based Volume Rendering C++ Code

B.1 vtkVolumeTextureMapper3DCg.h

```
...
class vtkVolumeTextureMapper3DCg : public vtkVolumeTextureMapper3D {
public:
    ...
    virtual void Initialize()
    virtual void Render(vtkRenderer* ren, vtkVolume* vol);
    ...
    static const float V_START_INDICES[24];
    static const float V_END_INDICES[24];
    static const float N_SEQUENCE[64];

protected:
    ...
    virtual void ComputePolygonsParameters(vtkRenderer* ren,
        vtkVolume* vol, float vertices[][3], float tCoordScale[3],
        float tCoordOffset[3], double plane[4], double &minDistance,
        double &stepSize, int &frontIndex);

    virtual void ComputePolygonsFP(vtkRenderer* ren, vtkVolume* vol);
    virtual void ComputePolygonsVPFP1(vtkRenderer* ren, vtkVolume* vol);
    virtual void ComputePolygonsVPFP2(vtkRenderer* ren, vtkVolume* vol);
    virtual void ComputePolygonsVPFP3(vtkRenderer* ren, vtkVolume* vol);
    virtual void ComputePolygonsVPFP4(vtkRenderer* ren, vtkVolume* vol);
    ...
    virtual int UpdateVolumes();
    virtual int UpdateColorLookup(vtkVolume* vol);
    virtual void RenderFP(vtkRenderer* ren, vtkVolume* vol);
    virtual void RenderVPFP(vtkRenderer* ren, vtkVolume* vol);
    void RenderOneIndependentNoShadeVPFP(vtkRenderer* ren,
        vtkVolume* vol);
```

```
    void SetupOneIndependentTextures(vtkVolume* vol);
    void Setup3DTextureParameters(vtkVolumeProperty* property);
    void RenderPolygons(vtkRenderer* ren, vtkVolume* vol);
    ...
private:
    ...
};
```

B.2 vtkVolumeTextureMapper3DCg.cxx

```
...
void vtkVolumeTextureMapper3DCg::Initialize()
{
    ...
    if (this->RenderMethod >= VPFP1_CG_METHOD)
    {
        if (this->RenderMethod == VPFP1_CG_METHOD)
            this->CGVertexProgram = cgCreateProgram(this->CGContext,
                CG_SOURCE, vtkVolumeTextureMapper3DCg_OneComponentBasicV,
                this->CGVertexProfile, "V_Passthrough", NULL);
        else if (this->RenderMethod == VPFP2_CG_METHOD)
            this->CGVertexProgram = cgCreateProgram(this->CGContext,
                CG_SOURCE, vtkVolumeTextureMapper3DCg_OneComponentV2,
                this->CGVertexProfile, "V_ComputingPolygons", NULL);
        else if (this->RenderMethod == VPFP3_CG_METHOD)
            this->CGVertexProgram = cgCreateProgram(this->CGContext,
                CG_SOURCE, vtkVolumeTextureMapper3DCg_OneComponentV3,
                this->CGVertexProfile, "V_ComputingPolygons", NULL);
        else if (this->RenderMethod == VPFP4_CG_METHOD)
            this->CGVertexProgram = cgCreateProgram(this->CGContext,
                CG_SOURCE, vtkVolumeTextureMapper3DCg_OneComponentV4,
                this->CGVertexProfile, "V_ComputingPolygons", NULL);
        ...
    }
    ...
    this->CGFragmentProgram = cgCreateProgram(this->CGContext,
        CG_SOURCE, vtkVolumeTextureMapper3DCg_OneComponentNoShadeF,
        this->CGFragmentProfile, "F_Sampling", NULL);
    ...
    if (this->RenderMethod == VPFP4_CG_METHOD)
    {
        CGparameter CGparameter_nSequence =
            cgGetNamedParameter(this->CGVertexProgram, "nSequence");
        cgGLSetParameterArray1f(CGparameter_nSequence, 0, 64,
            vtkVolumeTextureMapper3DCg::N_SEQUENCE);
        CGparameter CGparameter_v1 =
            cgGetNamedParameter(this->CGVertexProgram, "v1");
```

```
        cgGLSetParameterArray1f(CGparameter_v1, 0, 24,
            vtkVolumeTextureMapper3DCg::V_START_INDICES);
        CGparameter CGparameter_v2 =
            cgGetNamedParameter(this->CGVertexProgram, "v2");
        cgGLSetParameterArray1f(CGparameter_v2, 0, 24,
            vtkVolumeTextureMapper3DCg::V_END_INDICES);
    }
}

void vtkVolumeTextureMapper3DCg::Render(vtkRenderer* ren,
    vtkVolume* vol)
{
    ...
    if (this->RenderMethod == FP_CG_METHOD)
        this->RenderFP(ren, vol);
    else if (this->RenderMethod >= VPFP1_CG_METHOD)
        this->RenderVPFP(ren, vol);
    ...
}

void vtkVolumeTextureMapper3DCg::RenderVPFP(vtkRenderer* ren,
    vtkVolume* vol)
{
    glAlphaFunc(GL_GREATER, (GLclampf)0);
    glEnable(GL_ALPHA_TEST);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    int components = this->GetInput()->GetNumberOfScalarComponents();
    switch (components)
    {
    case 1:
        if (!vol->GetProperty()->GetShade())
            this->RenderOneIndependentNoShadeVPFP(ren, vol);
        break;
    default:
        break;
    }

    vtkgl::ActiveTextureARB(vtkgl::TEXTURE1_ARB);
    glDisable(GL_TEXTURE_2D);
    glDisable(vtkgl::TEXTURE_3D_EXT);

    vtkgl::ActiveTextureARB(vtkgl::TEXTURE0_ARB);
    glDisable(GL_TEXTURE_2D);
    glDisable(vtkgl::TEXTURE_3D_EXT);
}
```

```
void vtkVolumeTextureMapper3DCg::RenderOneIndependentNoShadeVPFP(
    vtkRenderer* ren, vtkVolume* vol)
{
    ...
    cgGLLoadProgram(this->CGVertexProgram);
    cgGLBindProgram(this->CGVertexProgram);

    CGparameter CGparameter_modelViewProj =
        cgGetNamedParameter(this->CGVertexProgram, "modelViewProj");
    cgGLSetStateMatrixParameter(CGparameter_modelViewProj,
        CG_GL_MODELVIEW_PROJECTION_MATRIX, CG_GL_MATRIX_IDENTITY);

    cgGLEnableProfile(this->CGVertexProfile);

    cgGLLoadProgram(this->CGFragmentProgram);
    cgGLBindProgram(this->CGFragmentProgram);
    cgGLEnableProfile(this->CGFragmentProfile);

    this->CGTex3DParameter_volume =
        cgGetNamedParameter(CGFragmentProgram, "volume");
    this->CGTex2DParameter_colorLookup =
        cgGetNamedParameter(CGFragmentProgram, "colorLookup");

    cgGLEnableTextureParameter(this->CGTex3DParameter_volume);
    cgGLEnableTextureParameter(this->CGTex2DParameter_colorLookup);

    this->SetupOneIndependentTextures(vol);
    this->RenderPolygons(ren, vol);

    cgGLDisableTextureParameter(this->CGTex3DParameter_volume);
    cgGLDisableTextureParameter(this->CGTex2DParameter_colorLookup);

    cgGLDisableProfile(this->CGFragmentProfile);
    cgGLDisableProfile(this->CGVertexProfile);
}

void vtkVolumeTextureMapper3DCg::RenderPolygons(vtkRenderer* ren,
    vtkVolume* vol)
{
    vtkRenderWindow* renWin = ren->GetRenderWindow();
    ...
    if (this->RenderMethod >= VPFP2_CG_METHOD)
    {
        if (this->RenderMethod == VPFP2_CG_METHOD)
            this->ComputePolygonsVPFP2(ren, vol);
        else if (this->RenderMethod == VPFP3_CG_METHOD)
            this->ComputePolygonsVPFP3(ren, vol);
```

```
        else if (this->RenderMethod == VPFP4_CG_METHOD)
            this->ComputePolygonsVPFP4(ren, vol);

        for (int i = this->NumberOfPolygons - 1; i >= 0; i--)
        {
            if (i % 64 == 1)
            {
                glFlush();
                glFinish();
            }
            if (renWin->CheckAbortStatus())
                return;
            glBegin(GL_POLYGON);
            for (int j = 0; j < 6; j++)
                glVertex2f(j, i);
            glEnd();
        }
    }
    else
    ...
}
```